BGSU Libraries
A11309872460

...OR SCOTT KNASTER

Macintosh Inside Out

System 7 REVEALED

ANTHONY MEADOW

System 7 Revealed

System 7 Revealed

Anthony Meadow

Addison-Wesley Publishing Company, Inc.
Reading, Massuchusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

BOWLING GREEN STATE UNIVERSITY
DISCARDED
LIBRARY
BOWLING GREEN STATE
UNIVERSITY LIBRARIES

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

Meadow, Anthony.
System 7 revealed / Anthony Meadow.
p. cm. — (Macintosh inside out)
Includes index.
ISBN 0-201-55040-7
1. Operating systems (Computers) 2. System 7. 3. Macintosh (Computer)—Programming. I. Title. II. Title: System seven revealed. III. Series.
QA76.76.063M43 1991 91-9681
005.265—dc20 CIP

Copyright © 1991 by Bear River Associates, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor: Carole McClendon
Technical Reviewer: Steve Goldberg
Cover Design: Ronn Campisi Design
Set in 10.5-point Palatino by Shepard Poorman Communications Corporation

2 3 4 5 6 7 8 9 10 - MW - 94939291
Second printing, July 1991

To Diana

Contents

Foreword by Scott Knaster xix

Acknowledgments xxi

Introduction xxiii
Why This Book? xxiii
An Overview of System 7 xxiv
Hands-on Outlines for Coding xxiv
A Guide for Future Reference xxiv
The Framework xxiv
A Brief Word on Terminology xxvii
Important Concepts of the Macintosh Operating System xxviii
Memory Management in the Macintosh Operating System xxviii
Resources xxix
QuickDraw xxxi
Events xxxii
Technical Documentation on the Macintosh Operating System xxxiii
Inside Macintosh xxxiv
Technical Notes xxxiv
Apple Technical Library xxxiv
APDA Documents and Software xxxv
Conclusion xxxv

1. **Overview of System 7** 1
Introduction 1
The Strategy behind System 7 1
The History of System 7 3

The Components of System 7 3
The Finder 4
The Human Interface 4
The Installer 5
Interapplication Communications and High-Level Events 5
The Edition Manager 6
TrueType and Fonts 6
TextEdit and International Services 7
The Data Access Manager 7
The Help Manager 8
The Sound Manager 9
The Communications Toolbox 9
AppleTalk and File Sharing 9
QuickDraw 10
The Memory Manager and Virtual Memory 11
Processes 11
The File System 12
The Hardware Managers 13
Running System 7 14
Memory Requirements 14
Virtual Memory Requirements 14
Conclusion 15

2. The Finder 17
Introduction 17
The Menu 17
The Apple Menu 18
The File Menu 18
The Edit Menu 18
The View Menu 19
The Label Menu 19
The Special Menu 20
Application Menu 20
Script Menu 21
Help Menu 21
The System Folder 21
The New System Folder 21
Font/DA Mover 22
Aliases 23
File Sharing 24
Desktop Objects 26
Navigating the Desktop 26
Stationery Pads 27

The Trash 27
Virtual Memory 27
Conclusion 28

3. **The Human Interface Guidelines** 29
Introduction 29
Color and Interface Design 29
Guidelines for Using Color in the Interface 30
Icons 30
New Kinds of Icons 31
Menus and Interface Design 32
New Standard Menu Commands 32
Pop-up Menus 32
Windows and Interface Design 33
Dialog Boxes and Interface Design 33
Movable Modal Dialog Boxes 34
Keyboard Navigation in Dialog Boxes 35
Changes to the Macintosh Human Interface Guidelines 35
Conclusion 36

4. **The Installer** 37
Introduction 37
Using the Installer 39
What the Installer is Really Doing 39
Developing an Installer Script 39
The Organization of an Installer Script 40
Installer Resources 42
The 'inrl' (Rule) Resource 42
The 'infr' (Rule Framework) Resource 44
Assertions 44
The 'inpk' (Package) Resource 44
The 'infa' (File Atom) Resource 45
The 'inra' (Resource Atom) Resource 45
The 'inaa' (Action Atom) Resource 46
The 'inat' (Audit Atom) Resource 46
The 'inbb' (Boot Block Atom) Resource 47
The 'icmt' (Installer Comment) Resource 47
The 'infs' (File Specification) Resource 47
The 'indo' (Disk Order) Resource 48
Customizing the Installer 48
Creating a Splash Screen 48
When to Use the Installer 48
Conclusion 49

5. **Compatibility** 51
Introduction 51
The Gestalt Manager 52
Calling Gestalt 52
Modifying the Gestalt Manager 55
Selector Functions 56
Running under A/UX 56
Availability of the Macintosh Operating System under A/UX 57
Programming Techniques for A/UX Compatibility 59
System 7-Aware Applications 59
Conclusion 61

6. **The PPC Toolbox, High-Level Events, and Apple Events** 63
Introduction 63
The PPC Toolbox 63
PPC Terminology 64
Naming PCC Ports 64
When to Use the PPC Toolbox, When to Use Apple Events 65
Compatibility and the PPC Toolbox 65
Managing PPC Services 65
Calling the PPC Toolbox 66
An Example of Using the PPC Toolbox 69
Using High-Level Events 71
Types of Events 71
Low-Level Events 71
Operating-System Events 72
High-Level Events 73
Describing High-Level Events 73
Calling High-Level Event Manager Routines 74
The 'SIZE' Resource 75
Apple Events 76
Compatibility and Apple Events 77
Important Data Structures of the Apple Event Manager 77
Using Apple Event Dispatch Tables 80
Extracting Data from an Apple Event 81
Sending an Apple Event 82
Where Are We Headed? 84
Conclusion 85

7. **The Edition Manager** 87
Introduction 87
The User's View of Publishing and Subscribing 88
Displaying Publishers and Subscribers 92

Changes to the Edit Menu 92
Supporting the Edition Manager from Various Types of Applications 93
The Edition File 94
Saving Documents That Contain Sections 94
Using the Edition Manager 94
Compatibility and the Edition Manager 95
Starting the Edition Manager 95
The Section Record 95
Working with Section Records 95
Creating a Publisher 96
Creating a Subscriber 97
Format Marks 98
Subscribing to Files Rather Than Editions 99
Apple Events and the Edition Manager 99
Implementing Edition Manager Support 100
Conclusion 100

8. Fonts and TrueType 103
Introduction 103
Macintosh Fonts: Then and Now 103
The First Fonts: 'FONT' Resources 104
PostScript Arrives 105
New Font Resources: 'NFNT' Resources 107
TrueType Emerges 108
The PostScript Controversy 110
TrueType Font Technology 110
Taking Full Advantage of TrueType 115
Using the Font Manager 115
How the Font Manager Chooses a Font 115
Compatibility and the Font Manager 116
Using the New Font Manager Calls 116
Conclusion 118

9. TextEdit and International Services 121
Introduction 121
Why the Script Manager? 121
Improvements in TextEdit 122
Compatibility and TextEdit 123
Script Manager Support in TextEdit 123
Using the New TextEdit Routines 124
Improvements in the Script Manager 128
Using the Script Manager 132

Compatibility and the Script Manager 136
Improvements to International Features 136
The International Utilities Package 136
The International Resources 137
The 'itlc' Resource 138
The 'itlb' Resource 138
The 'itlm' Resource 139
The 'itl0' Resource 139
The 'itl1' Resource 139
The 'itl2' Resource 140
The 'itl4' Resource 140
The 'KCHR' Resource 141
The 'kcs#', 'kcs4', and 'kcs8' Resources 141
The 'KCAP' Resource 141
The 'KSWP' Resource 142
The 'itlk' Resource 142
Writing International Software 142
Conclusion 143

10. The Data Access Manager and the Data Access Language 145
Introduction 145
Why the Data Access Language? 146
The Architecture of the Data Access Manager 148
Queries and the Data Access Manager 150
What Do You Need to Run DAL? 150
The Data Access Language 151
Aspects of Using DAL 154
Security Considerations 156
Examples of Data Access Language 157
When to Use DAL and When to Use a DBMS Directly 158
The Future of the Data Access Language 160
Using the Data Access Manager 160
Compatibility and the Data Access Manager 161
Query Documents 161
Using the High-Level Calls to the Data Access Manager 162
Using the Low-Level Calls to the Data Access Manager 163
Using Result Handlers 164
Using the Low-Level Utility Calls 165
Human Interface Guidelines and the Data Access Manager 166
Conclusion 167

11. The Help Manager 169
Introduction 169
Amount of Help versus Amount of Work 171

Compatibility Considerations 172
Internationalization Considerations 172
Hot Rectangles! 172
Help, the State, and Help Messages 173
Help Resources 174
The 'hmnu' Resource 175
The 'hdlg' Resource 176
The 'hrct' Resource 177
The 'hwin' Resource 178
The 'hfdr' Resource 179
The 'hovr' Resource 179
Creating Help Resources for Standard Menus 180
Creating Help Resources for Modal Dialogs 181
Creating Help Resources for Modeless Dialogs and Windows 181
Creating Help Resources for Custom Menus 182
Creating Help Resources for Movable Window Objects 183
Using the Help Manager Routines 183
Example: Creating an 'hmnu' Resource 184
Writing Help Messages 185
Conclusion 186

12. The Sound Manager 187
Introduction 187
Different Ways to Produce Sounds 188
Sound Data Structures 189
Sound Resources 189
Sound Files 189
Calling the Sound Manager at a High Level 190
Calling SysBeep 190
Producing Sounds the Easy Way with SndPlay 191
Using SndStartFilePlay to Play Sounds on Disk 191
Architecture of the Sound Manager 191
Applications and Sound Channels 192
Modifiers 193
Synthesizers and 'snth' Resources 193
Sound Hardware 195
Limitations 195
Sound Commands 196
Calling the Sound Manager at a Low Level 198
Managing Channels 198
SndDoCommand and SndDoImmediate 198
Playing Notes and Installing Instruments 199
Playing Sampled Sounds 199

Playing Sampled Sounds from Disk 199
Managing Sound Channels 200
Managing Channel Capacity 200
Compressing and Expanding Sound 201
Compatibility and the Sound Manager 201
Getting Sound Status Information 202
Installing Modifiers 203
Recording Sounds 203
Sound Input and Its Implications 203
Sound Input Devices 204
Using the High-Level Interface to Record Sounds 205
Using the Low-Level Interface to Record Sounds 206
Conclusion 207

13. The Communications Toolbox 209
Introduction 209
The Architecture of the Communications Toolbox 209
Data Structures 211
The Communications Toolbox in the Larger Picture 212
Compatibility and the Communications Toolbox 212
Programming with the Connection Manager 213
The Connection Record 215
Using the Connection Manager Routines 215
Preparing to Use a Connection 216
Opening, Managing, and Closing Connections 217
Reading, Writing, and Searching over a Connection 218
Handling Events with the Connection Manager Routines 218
Other Connection Manager Routines 219
Internationalization and the Connection Manager 219
Writing a Connection Tool 219
Creating a 'cdef' Resource 221
Creating a 'cval' Resource 223
Creating a 'cset' Resource 223
Creating a 'cscr' Resource 223
Creating a 'cloc' Resource 224
Creating a 'cbnd' Resource 224
Programming with the Terminal Manager 224
The Terminal Record 227
Using the Terminal Manager Routines 228
Preparing to Emulate a Terminal 228
Emulating a Terminal 229
Working with Special Keys 230

Handling Events with the Terminal Manager Routines 230
Other Terminal Manager Routines 230
Internationalization and the Terminal Manager 231
Writing a Terminal (Emulation) Tool 231
Creating a 'tdef' Resource 232
Programming with the File Transfer Manager 234
The File Transfer Record 235
Using the File Transfer Manager Routines 236
Preparing to Transfer a File 236
Performing the File Transfer 237
Handling Events with the File Transfer Manager Routines 237
Other File Transfer Manager Routines 238
Internationalization and the File Transfer Manager 238
Writing a File Transfer Tool 238
Creating an 'fdef' Resource 239
Programming with the Communications Resource Manager 240
Programming with the Communications Toolbox Utilities 241
Conclusion 242

14. AppleTalk Phase II and AppleShare 245
Introduction 245
AppleTalk Protocols and Drivers 246
Protocols of Interest to Application Programmers 248
AppleTalk Phase II 248
AppleTalk Phase II Features for Application Developers 249
Improvements in the AppleTalk Transaction Protocol 250
Improvements to the Zone Information Protocol 250
The AppleTalk Transition Queue 250
Using the AppleTalk Transition Queue 251
Obtaining Current Node Information Using the .MPP Driver 252
Wildcard Characters for the Name Binding Protocol 252
Compatibility and AppleTalk 253
AppleTalk Data Stream Protocol 253
Using the ADSP—Important Data Structures 253
The ADSP Calls 254
Using the ADSP Calls 255
The ADSP Calls for a Connection Listener 256
File Sharing 257
Conclusion 259

15. QuickDraw 261
Introduction 261
Some Background on QuickDraw 261

32-bit QuickDraw 262
Direct Color Devices 262
Indexed Color Devices 263
Improvements in 32-bit QuickDraw 263
Support for Direct Color in PixMaps 263
Support for Direct Color in the PICT2 Format 265
Using New QuickDraw Calls 267
Improved Gray-Scale Support 268
Telling QuickDraw That You've Been Fiddling 268
The Color Picker Package 269
Converting between Color Models 270
The Palette Manager 270
Types of Colors 270
Creating and Using 'pltt' (Palette) Resources 271
Using Palettes with Windows 272
Working with Palettes 273
Animating Palettes 273
The Graphics Device Manager 274
Creating GWorlds 274
Using GDevice Records 275
The Picture Utilities Package 277
Examining PICT Files 277
Examining Pixel Maps 278
Customizing the Picture Utilities 279
Conclusion 279

16. The Memory Manager 281
Introduction 281
Virtual Memory 281
Compatibility and Virtual Memory 283
Controlling Virtual Memory 284
So Who Does Have to Worry about Virtual Memory? 285
Temporary Memory 286
Differences in Temporary Memory: Now and Then 287
Compatibility and Temporary Memory 287
Temporary Memory Calls 287
32-Bit Cleanliness 289
WDEFs, CDEFs, and Cleanliness 291
Calculations on Memory Addresses 291
Passing Along the A5 World 292
Compatibility and 32-Bit Cleanliness 292
Conclusion 292

17. Processes 295
Introduction 295
The Process Manager 296
Process Serial Numbers 296
Process Scheduling and Switching 296
Launching Applications 298
Launching Desk Accessories 299
Getting Information about Other Processes 299
Compatibility and the Process Manager 301
The Notification Manager 301
Using the Notification Manager 302
The Time Manager 302
Accuracy of the Time Manager 303
Fixed-Frequency Scheduling 304
Time Manager Tasks 304
Compatibility and the Time Manager 304
Creating and Using Time Manager Tasks 305
Measuring Time Intervals 305
Conclusion 306

18. The File System 307
Introduction 307
Compatibility and the File and Alias Managers 308
File IDs 308
Working with File IDs 309
FSSpec Records 310
Searching for Files 310
Other Changes to the File Manager 311
Compatibility and the File Manager 313
Changes to the Resource Manager 313
Aliases 313
Alias Records, Canonical File Specifications 314
Calling the Alias Manager 315
The Standard File Package 316
Compatibility and the Standard File Package 319
Finder Information and the Desktop Database 319
System Calls to Access the Desktop Database 320
New Icon Types Supported by the Finder 321
Document String Resources Supported by the Finder 321
Architecture of the System Folder 322
Locating the Preferences and Temporary Folders 323
Conclusion 324

19. The Hardware Managers 325
Introduction 325
The Slot Manager 325
Slot Manager and the Initialization Process 326
Compatibility and the Slot Manager 327
New Slot Manager Routines 327
The Power Manager 327
Conclusion 329

20. Object-Oriented Programming and System 7 331
Introduction 331
Object-Oriented Programming 331
What Is MacApp? 334
But What about Performance and Memory? 335
When to Use MacApp 336
When Not to Use MacApp 336
Resources for MacApp Programmers 337
Conclusion 338

21. The Future 339
Introduction 339
Some Future Features 339
New Print Architecture 340
File System Manager 340
Layout Manager 340
Revised SCSI Manager 340
Definite Futures 341
AppleScript 341
Apple Event Object Model 341
Memory Protection and Multiple Address Spaces 341
Preemptive Multitasking 341
Multimedia Support 342
Network Booting 342
Indefinite Futures 342
Revised QuickDraw 343
Object-Oriented Programming 343
Conclusion 343

Index 345

Foreword by Scott Knaster

The creation of Macintosh System 7.0 was the largest software effort ever undertaken by the good folks at Apple. Scores of software engineers invested hundreds of person-years and untold megagallons of Jolt Cola to get the thing written; dozens of quality engineers made sure it all worked; flocks of product managers got the world ready for it; myriads of other Apple people of all types did what they did; handfuls of writers wrote about it; and now, zillions of developers get to make great applications out of it. The mind reels.

What does it all mean? Part of the System 7 effort was writing stuff to help interested people find out why System 7 is important to them. The most important technical documentation that comes from Apple is *Inside Macintosh Part 6: Freddy Krueger Burns the ROM*. Whoops, make that *Inside Macintosh*, Volume VI, of course, the definitive reference for how to use Macintosh system software. Thus it has ever been, as ever it shall be.

Inside Macintosh, Volume VI does a wonderful job of giving you all the details of how to use the new Toolbox Managers and system calls. The software is so complex, though, that there's room for more than one kind of presentation of this stuff. In this book, Tony Meadow gives you a very different view from the one presented in *Inside Macintosh*.

Instead of explaining every call in great detail, Tony introduces you to each lovely part of System 7 and tells you lots about it. His gentle descriptions of each Toolbox Manager inform you about what that Manager does, why it was created, how you might take advantage of it, and what lies in store for your users when you do. Tony is enthusiastic about System 7's riches because he's a developer and he understands that it's a nice

present from Apple, but he's also realistic in letting you know what might get you into trouble (watch out for that low ceiling ahead!).

After reading this book, you should be System 7-savvy and all ready to dive into writing your application with *Inside Macintosh* by your side. Have a good time as you utilize all the hard work by all those good Apple citizens, and take good advantage of the insights that Tony presents in this book.

Scott Knaster
Macintosh Inside Out Series Editor

Acknowledgments

Many people helped make this book possible. First, I'd like to thank the programmers, testers, writers, the product marketing team, the managers, the many people who spoke at the 1989 and 1990 Apple Worldwide Developers Conferences and all the other folks at Apple who have been working on System 7. You did good—it's great stuff!

Quite a few people at Apple helped make this book possible, including Steve Goldberg, Charlie Oppenheimer, Michael Wallace, Sharon Everson and the Tech Pubs group, Diana Mason, Martha Steffen, Bobby Carp, Phil Goldman, Jon Magill, Chris Derossi, Gregg Williams, Steve Friedrich, and the MacApp team

Carole McClendon, Joanne Clapp Fullagar, and Rachel Guichard in the Berkeley Addison-Wesley office got me to write this book, developed it, and handled all the details, respectively. Thanks for all your help. Tanya Kucak did a splendid job of copy editing. Mary Cavaliere and the production team and Abby Cooper on the East coast produced the book and got it to market. Thank you.

Thanks to the gang at Bear River Associates and Bear River Institute who helped by giving me some time during the day to work on the book and some feedback: Randy Matamoros, Dave Richey, Steve Evangelou, Mark Hall, John Wilkinson, Sharon Ryals, Ant Bonét, and Robin Belkin.

Scott Knaster suggested the idea of this book. Dave Wilson and Paul Hoffman were very encouraging when I started this project. Michael Wallace, Michael Peirce, and Jeff Cherniss took over the MacSEF group from me, allowing me more time for this book. Jim Von Ehr and Tom Irby, both from Altsys, graciously provided the illustrations in the TrueType chapter. Thank you.

Thanks also to Mark Davis (Apple), Jim Von Ehr (Altsys), and Jim Stoneham (Apple) for reviewing parts of the book. I'd especially like to thank Steve Goldberg (Apple), Yogen Dalal (Claris), and Art Schumer (Microsoft) for reviewing the entire book. Any remaining errors are, of course, my fault.

Last, I'd like to thank Diana, Jeremy, and even Erica for the many evenings and weekends that allowed me to write this book.

Introduction

System 7 is the biggest change in the Macintosh since this family of computers was first introduced in 1984. Apple's technical documentation on System 7 as presented in *Inside Macintosh,* Volume VI, is approximately 1700 pages, and not all of System 7 is documented there. Important System 7 subjects, such as the Communications Toolbox Manager, the Data Access Language, the TrueType font format, and the one-button Installer are described in separate pieces of documentation. If you added up all the documentation on all the components of System 7, you'd be looking at almost 2500 pages.

▶ Why This Book?

This book is oriented primarily to Macintosh programmers who want to learn System 7. I assume that you have prior knowledge of the Macintosh operating system, and that you are now confronting the task of understanding just what System 7 is. This job isn't for the faint of heart.

Network managers, system administrators, and programmers from platforms other than the Macintosh will also find useful information in this book. For those readers, the Introduction presents a brief overview of Macintosh terms and concepts. The concepts that form the basis of each portion of the operating system are discussed in that chapter. For the most part, all you need to understand these concepts is a basic understanding of operating systems and familiarity with at least one programming language.

In these pages, you will find what you need to understand System 7 and to begin programming with it. Here is what you will find.

▶ An Overview of System 7

This book includes a clear and succinct overview and summary of each new aspect of the Macintosh operating system, which will give you a snapshot of System 7 as a whole. When you have understood what all of the components are, you will be ready to move on to *Inside Macintosh*, Volume VI, for more detail. The biggest problem in understanding System 7 is that there is no grand new architecture. For a start, Apple has improved many of the existing portions of the operating system. Apple has then gone on to redefine what is system software and what is application software. By radically raising the level of what belongs to system software, it is gaining an edge over other system vendors. Examples of this new level of system software include high-level toolkits, such as the Data Access Manager and the Communications Toolkit.

▶ Hands-on Outlines for Coding

Beyond explaining each aspect of System 7, this book provides outlines that should serve you when you need to accomplish a specific task using that portion of the operating system. Each outline explains which system calls to use and the order in which to use them. For specific code examples, look at Volume VI of *Inside Macintosh*. Unlike previous volumes, there are many examples in it. The Developer Technical Support group at Apple will also release additional examples that will be widely available.

▶ A Guide for Future Reference

After reading this book, you will have a good understanding of System 7 and, in particular, you'll have seen the important new system calls. Later, when you begin adding support for System 7 features into your own application and using Apple documentation, you can use this book as a roadmap. Each chapter ends with a section titled "Get Info," where you can look for references to the documentation on that aspect of the operating system. Remember that not all of System 7 is described in *Inside Macintosh*.

▶ The Framework

Each chapter describes the changes to one aspect of the Macintosh operating system. The chapters progress from those aspects of System 7 that are most visible to users to those that are the least visible. Not everything fits smoothly into this progression, but it does give you an idea where to look.

Chapter 1, "Overview of System 7," reviews the history of and Apple's strategy behind System 7. You'll also read succinct descriptions of each

new (or changed) component of the operating system. The chapter also describes the requirements for running System 7.

Chapter 2, "The Finder," explains how the behavior of the Finder has changed. Remarkably, the Finder has been completely rewritten for System 7 and implements an improved interface, but it still looks much like earlier versions of the Finder.

Chapter 3, "The Human Interface Guidelines," provides you with a description of how the Macintosh human interface guidelines have evolved. Some of the new features have a significant impact on the user interface, and Apple has been working hard to provide some guidance on what works best.

Chapter 4, "The Installer," explains the new one-button Installer. If your software package requires a variety of files and resources to be installed in a variety of locations, consider using the new Installer. This version of the Installer uses rules that you provide to simplify the installation of complex packages.

Chapter 5, "Compatibility," provides you with an overview of how to write software that will remain compatible with new versions of the operating system and with other environments such as A/UX, Apple's version of the UNIX operating system. You'll also learn about the **Gestalt** call, which provides your applications with a standard and safe way to get the parameters of the operating system, such as which version of QuickDraw is running.

Chapter 6, "The PPC Toolbox, High-Level Events, and Apple Events," explains the interapplication communications facilities of the Macintosh operating system. These are new with System 7 and will provide some exciting new possibilities for providing more features in applications without having to write all the code.

Chapter 7, "The Edition Manager," explains a new mechanism for sharing data between applications. Similar to cut and paste, the Edition Manager provides powerful new features. Once the user has subscribed to an edition, the Edition Manager ensures that the latest version of the edition is used whenever the user edits or prints the document.

Chapter 8, "Fonts and TrueType," gives you an overview of the new TrueType font architecture. It provides high-quality outline fonts that can be used both on the screen and in printing, improving the correspondence between screen and paper. Applications don't have to do much to use TrueType, but the new Font Manager calls that go along with it are explained.

Chapter 9, "TextEdit and International Services," describes the improvements made to TextEdit, which provides the basic text-editing facilities in the operating system. You'll also read about the improvements to the Script Manager and the International Utilities Package, which

together make it easy for your applications to be used in other languages around the world.

Chapter 10, "The Data Access Manager and the Data Access Language," outlines the Data Access Manager. This new manager provides a generic programming interface to database management systems and databases. In System 7, it is accompanied by the Data Access Language, which provides a powerful programming language encompassing an SQL-like set of commands for accessing data in relational database management systems.

Chapter 11, "The Help Manager," introduces this new system-level help facility. For the most part, you can provide help for users of your applications by simply adding some resources to the application file. In some cases, you may have to write a small amount of code to support this help feature.

Chapter 12, "The Sound Manager," presents the changes made to the Sound Manager. This latest version makes it easier to support multiple sound channels and play sounds from disk. The Sound Manager also provides a new set of routines for recording sounds.

Chapter 13, "The Communications Toolbox," details this new toolbox, which provides a standard programming interface for three basic communications functions: connection management, terminal emulation, and file transfer.

Chapter 14, "AppleTalk Phase II and AppleShare," reviews AppleTalk Phase II, which becomes a standard part of the operating system with this release. The most important aspect for application programmers is that the AppleTalk Data Stream Protocol is part of the standard operating system. This protocol is easy to use and provides almost all the features that application programmers (as opposed to system-level programmers) require for supporting multiuser applications. File Sharing is also described. This software provides a subset of the AppleShare file server software usable by any System 7 user.

Chapter 15, "QuickDraw," describes 32-bit QuickDraw, the latest version of QuickDraw, which also becomes a standard part of the operating system with System 7. The new version provides support for 32-bit direct color devices and performance improvements in general.

Chapter 16, "The Memory Manager," provides details on two changes in the Memory Manager: virtual memory and temporary memory. Virtual memory allows a user to substitute disk space for more expensive RAM, which enables users to run more applications at a time than would otherwise be possible. It does not, for the most part, directly affect any applications. Temporary memory, the unallocated memory used for launching applications, is available for temporary use by other applications under less restrictive conditions than before.

Chapter 17, "Processes," discusses several managers related to managing processes in the operating system. The Process Manager, new with System 7, provides routines for launching applications and desk accessories, and for getting the status of any currently running process. As yet, the Macintosh operating system does not support preemptive multitasking, but this manager does give an indication of where the operating system is heading. The Notification Manager, which provides notification services for software running in the background, is discussed. The latest version of the Time Manager, which provides time measurement services and accurate scheduling of small tasks under its control is also described.

Chapter 18, "The File System," details the many changes that System 7 brings to the file system of the Macintosh. The major pieces are file IDs, FSSpec records, the new **PBCatSearch** routine for fast searches through a volume, aliases that allow a file to be in more than one place at a time, improvements to the File Manager, improvements to the Standard File Package, access to Finder information such as file icons and the "Get Info..." comments, new Finder icons, and the new architecture of the System Folder.

Chapter 19, "The Hardware Managers," looks briefly at the Slot Manager and the Power Manager. The Slot Manager manages the NuBus cards in the Macintosh II family. The Power Manager handles the electrical power of the Macintosh Portable, which is always on. Neither of these managers will be of interest to most application programmers.

Chapter 20, "Object-Oriented Programming and System 7," discusses the benefits of using object-oriented programming with Apple's MacApp application framework. MacApp programmers are getting some of the improvements that System 7 brings with little work. As the operating system increases in complexity (and power), object-oriented programming techniques make it easier for developers to create software.

Chapter 21, "The Future," describes some of the changes that developers can expect to see as the Macintosh operating system continues to evolve. Some of these changes have been described by Apple, but others are changes that we should expect to see in any case.

▶ A Brief Word on Terminology

Let's pause to review basic Macintosh terminology. If you're not a Macintosh programmer, read this section before proceeding. If you're already a Macintosh programmer, you are undoubtably familiar with these terms, so you can skip this section.

The Macintosh operating system does not have a name, such as MS-DOS or UNIX. Perhaps this is for historical reasons—Apple has tried from the beginning to reduce the amount of jargon used in software and manuals. Actually, you might think of the operating system as divided

into two layers: the operating system proper and the Toolbox. Both the operating system and the Toolbox are organized as a set of *managers*. Each manager implements some functions. For example, the Memory Manager, part of the operating system, provides a set of *system calls*, or *routines*, to allocate and deallocate blocks of memory. (This manager provides other functions besides these two.)

The operating system provides the usual kinds of services that most other operating systems do, as in the following list:

- Managing memory
- Process management (yes, the Macintosh does support limited forms of multiprocessing)
- Managing input/output devices (such as disks, keyboard, and mouse)
- Providing file services.

Several dozen managers make up the Macintosh operating system.

The Toolbox implements the Macintosh human interface. The distinction between the operating system and the Toolbox is a concept that is rarely mentioned in Apple's technical documentation. This distinction is not important in practice, either; few programmers worry about whether a manager is part of the operating system or the Toolbox. In this book, the term *operating system* will be used indiscriminately to mean both the operating system proper and the Toolbox.

▶ Important Concepts of the Macintosh Operating System

As mentioned earlier, the Macintosh operating system provides most of the same kinds of services as most other operating systems (such as UNIX or VAX/VMS). However, several concepts are quite different from other operating systems: memory management, resources, QuickDraw, and events.

▶ Memory Management in the Macintosh Operating System

Every operating system controls the use of memory by software. With most operating systems, software allocates memory by asking for a block of a specified size from the heap. The *heap* is a chunk of memory, blocks of which can be allocated from anywhere in the heap. This is unlike the

stack, in which memory must be allocated starting with the next byte on the stack. Stacks are useful when memory will be allocated in sequence and then deallocated in the reverse order. A heap is useful when memory must be allocated in a nonsequential manner.

If the system call to allocate memory from the heap is successful, it returns a pointer to the block of memory. Macintosh software can allocate memory in this fashion, but the preferred method is to ask for a handle to the block rather than a pointer. A *handle* is a pointer to a pointer.

Why use this double level of indirection? When the Macintosh operating system was first released, handles provided a method for the operating system to have more control over memory than if all memory were allocated using pointers. Since the application uses only the first pointer, the operating system is free to move the block of memory to which the handle points. This means that if a large block of memory is requested, but no contiguous block of that size is available, the operating system can try to make a block of that size available by shifting handle blocks around in memory. If sufficent blocks can be shifted, then the request for the large block can be fulfilled.

Clearly, by encouraging software to use handle blocks, rather than pointer blocks, the operating system has more flexibility in managing memory. Thus, programmers don't have to worry about memory allocation as much as they would have to otherwise.

As a consequence of this memory management scheme, most applications have a large number of small code segments. This allows the operating system to move segments in and out of memory as the memory requirements of the application change. Applications typically require more memory when they start up, open a document, or print a document.

Virtual memory, available to some Macintoshes under System 7, will not change the way applications use memory. Applications will continue to use handles in preference to pointers, because many Macintosh systems do not support virtual memory. You could write an application that required virtual memory, but the potential market for it would be a lot smaller.

▶ Resources

Resources are an operating system concept that started on the Macintosh. They provide a powerful capability for separating data from code, and more generally, providing templates for complex data structures. Resources are easily created and manipulated in software.

A *resource* is a block of data identified by a resource type, which is a 4-byte code; a resource ID, which is an 16-bit integer; and, optionally, a

name. Resources are ubiquitous in Macintosh software. For example, an application has many 'CODE' resources; each code segment is contained in one of these resources. The icons displayed by the Finder are each a resource. An 'ICN#' resource contains the standard 32-bit by 32-bit black-and-white icon displayed by the Finder. Fonts and their bitmaps are described by 'FOND', 'NFNT', and 'sfnt' resources. Cursors are described by 'CURS' resources. Strings are described by 'STR ' resources, and a set of strings is described by an 'STR#' resource.

The operating system and Toolbox use many resources. Applications can create their own resource types. A resource can support almost any structure for the data contained within it. If the resource type is proprietary, however, only applications that know about that resource type can use them.

Resources are used throughout the Macintosh operating system. They have had a particularly large effect in the File Manager. Files on most operating systems are simply contiguous, finite strings of bytes. Files on the Macintosh are like this, too. That is, the *data fork* of a Macintosh file is like that, but Macintosh files have *two* forks. The *resource fork* of a Macintosh file contains resources. The structure of the resource fork is not especially well documented because software should access the resources in a resource fork through the routines of the Resource Manager. Note that because Macintosh files have two forks, transferring files to other types of machines (running MS-DOS or UNIX, for example) is a more complex problem than if the files had only a single fork.

Key Point ▶ For the most part, application files have little (if anything) in the data fork. Almost all of the application resides in the resource fork. Data files, on the other hand, are almost all data fork and very little resource fork. Note that these are generalizations.

There are several restrictions with respect to resources. First, the resource fork cannot be shared by multiple processes. If data needs to be shared (over a network, for example), then it must be stored in a data fork, not a resource fork. Second, do not redefine any standard resource types. If you are creating a new resource, make sure that the resource type does not conflict with any of the standard resource types. *Inside Macintosh* and some of the Tech Notes list the standard resource types.

Do not use the resource fork of a file as a database unless it will contain only a small number of items (a couple of hundred or fewer). This is because the structure of the resource fork isn't optimized for use as a

high-performance database. If you need to use a database, then either write one yourself or buy someone else's. If you use a resource fork as a database with thousands of entries, you will have performance problems. Also, the resource fork of a file cannot be shared across a network, unlike the data fork.

By the Way ▶

How did resources come about? The Macintosh folklore is that they were invented as a result of Pascal. Apple, because of the heavy use of UCSD Pascal on the Apple II, the Apple III, and the Lisa, used Pascal to write a lot of Macintosh operating system. Pascal does not support static initialization (as the C language does). Resources were invented as a way to initialize large data structures without having to write a lot of code.

Once the concept was there, resources were used all over the operating system because they are such a natural concept. You will see many new resource types described in this book. The Installer, the Help Manager, and the text services in System 7 all use resources to perform their functions.

Resources have been the key to the internationalization of the Macintosh operating system and application software, beginning with fonts. The Macintosh was the first affordable machine that supported multiple fonts as a basic capability. Soon afterward, many people created fonts for other languages, such as French, Greek, and even Sanskrit. Apple eventually created the Script Manager as a component of the operating system to provide support for languages that need more than one or two hundred characters, such as Chinese, Japanese, and Korean. The Script Manager also supports languages that are written from right to left, such as Hebrew, Arabic, and Farsi. The Script Manager uses a lot of resources to accomplish its functions. For more on the Script Manager, see Chapter 9.

▶ QuickDraw

QuickDraw can be thought of as the graphics manager of the Macintosh operating system. Everything that appears on a Macintosh screen or on a printer was drawn by the QuickDraw routines. (Actually, that last statement isn't completely true—advanced programmers sometimes print using PostScript code directly.)

QuickDraw provides data structures for basic kinds of graphic objects, including lines, rectangles, rounded rectangles, ovals, wedges, polygons, and

regions. A *region* is an arbitrarily shaped object described as a collection of the other types of shapes. Regions are especially important because you can easily manipulate a complex graphic with QuickDraw routines.

QuickDraw provides a set of data structures and a set of routines that work with them. The QuickDraw routines are used throughout the operating system and applications to draw all graphics. All of the QuickDraw operations are ultimately implemented by a set of low-level routines known as the QuickDraw bottleneck procedures. This set of 13 routines varies from device to device, because drawing on a display is different from drawing on a printer.

The QuickDraw bottleneck procedures dramatically reduce the support for various displays, printers, and other output devices. Traditionally, operating systems know nothing (or very little) about output devices. Therefore, the application programmer must write code to support each type of display and each type of printer. Word processing programs under MS-DOS typically require a lot of code to support the hundreds of different types of printers and the dozens of types of displays. The Macintosh operating system is different because application programmers do not have direct access to displays or printers. All drawing goes through QuickDraw. The primary consequence of this is that the display or printer vendor provides the operating-system-level support for its output device. Another consequence is that most applications are almost completely device-independent with respect to both displays and printers.

Macintosh applications are rarely device dependent. Some types of applications, such as color paint programs, can be used only on color displays. Some high-end publishing applications use PostScript and consequently must be used with a PostScript printer.

▶ Events

Events are generated either by the user or by the operating system. The user generates events whenever a key is pressed down or released, whenever the mouse button is pressed down or released, or whenever a disk is inserted into a drive. Note that moving the mouse does not create an event. The operating system also generates events whenever the relative order of windows is changed, such as when a user clicks on a window behind the active window.

Macintosh applications are event-based. This means that each application uses the following basic structure:

1. Initialize the application
2. Get an event

3. Process it
4. Loop
5. Close the application

This is a very different structure than applications written for UNIX or MS-DOS. Under these other operating systems, applications typically prompt the user and wait for a response. The code that communicates with the user need not be structured in any particular way.

Because the Macintosh operating system encourages event-based applications, Macintosh applications typically appear to be controlled by the user. Under MS-DOS and other operating systems, the application typically appears to be in control. The Macintosh human interface guidelines require that the user be in control as much as possible. The operating system supports this directly.

System 7 extends the concept of events. For the most part, events before System 7 are a low-level concept. A key-down event does not carry much meaning. It must be extensively interpreted to provide meaning to the application. The concept of high-level events has been added with System 7. A high-level event might be "print this document"—obviously a much higher-level concept than a key-down event. As you will see later in this book, high-level events will lead to interesting and powerful changes in the way that you interact with applications and documents.

▶ Technical Documentation on the Macintosh Operating System

Before we move onto other topics, let's briefly look at the technical documentation on the Macintosh operating system. Apple provides four sets of documentation. Let's first look at these.

Note ▶

Apple is different from most other computer companies—they still have a sense of humor. Bits of humor crop up in the documentation, but you'll see them only if you look for them. As an example, one of the Time Manager routines is named **PrimeTime**. Another example is the names for the Developer CDs, sent once every quarter to all Apple Associates and Partners. Some of the titles are "Phil and Dave's Excellent CDs," "Night of the Living Disk," and "A Disk Named Wanda."

▶ *Inside Macintosh*

The most important set of technical documentation is *Inside Macintosh,* which was originally published as a three-volume set. With the introduction of each significant new member of the Macintosh family of computers, a new volume was added. Volume IV was written to describe the changes to the Macintosh operating system made with the Macintosh Plus, primarily the new Hierarchical File System (HFS). Volume V includes all the changes to the Macintosh operating system made for the Macintosh II, the first modular Macintosh. Among the many topics covered there are Color QuickDraw and the Slot Manager, which provides access to NuBus cards. Volume VI of *Inside Macintosh* was written to describe all the changes to the operating system brought by System 7. Volume VI is very large—for good reason, as you will see. Every volume of *Inside Macintosh* should be on every Macintosh programmer's shelf—you can't program the Macintosh without it.

▶ Technical Notes

Another important set of documentation is known as the Technical Notes. These notes, distributed by Apple to members of the Apple Partners and Associates program, are available to anyone else by subscription through APDA, the Apple Programmer's and Developer's Association, and are also widely distributed to most electronic information systems. Tech Notes can be freely copied, but cannot be sold. The Tech Notes describe miscellaneous features of the operating system that didn't make it into *Inside Macintosh,* techniques for using the operating system that are not documented anywhere else, bugs in the operating system, bugs in Apple's development tools, and so on. Every programmer should be aware that they exist and read through the list of titles regularly. About three hundred Tech Notes are out now.

It is often difficult, especially for programmers new to the Macintosh, to find their way around *Inside Macintosh* and the Tech Notes. This is because both sets of documentation have evolved over a five-year period. Apple will reorganize them.

▶ Apple Technical Library

A third set of technical documentation is the Apple Technical Library, published by Addison-Wesley. These books document relatively stable aspects of both the Macintosh software and hardware. They are all writ-

ten by technical writers at Apple. This series includes the following volumes:

- All the volumes of *Inside Macintosh*
- *Inside AppleTalk,* which describes the AppleTalk protocol suite
- *Human Interface Guidelines: The Apple Desktop Interface,* which documents the Macintosh human interface guidelines (every programmer should have a copy of this book)

▶ APDA Documents and Software

The last set of technical documentation is not really a set. It consists of the documents and software available through APDA, the Apple Programmer's and Developer's Association. This department at Apple distributes all technical documents and a great deal of software, such as Apple's own development tools. By joining APDA, which requires signing an agreement and paying a small annual membership fee, you can also purchase beta versions of some software and documentation. Among the many products available from APDA are the following:

- *Software Applications in a Shared Environment*—Documents how to write multiuser software using file servers (such as the new File Sharing feature of System 7)
- *Macintosh Programmers Workshop (MPW)*—Apple's own development system; also available are compilers for Pascal, C, and C++
- *MacApp*—Apple's primary object-oriented application framework, which you can use with either Pascal or C++; it includes full source code

▶ Conclusion

In this chapter, you've looked at how this book is structured, as well as how to use this book in conjunction with *Inside Macintosh,* Volume VI, and other Apple technical documentation. Readers who are less familiar with the Macintosh have had a chance to review terminology and major concepts of the Macintosh operating system.

Get Info ▶

Inside Macintosh, Volume VI, documents most of what is new in System 7. This book is published for Apple Computer by Addison-Wesley and is available through any bookstore. If you are programming for System 7, you will need this book by your side. You will need the previous volumes of *Inside Macintosh* as well if you are programming the Macintosh.

If you are new to the Macintosh, you should also look for some introductory Macintosh programming books, such as *Macintosh Programming Primer*, Vols. I and II (Addison-Wesley 1989, 1990). Another resource is Stephen Chernicoff's four-volume set entitled *Macintosh Revealed* published by Hayden Books. More experienced programmers will also be interested in other books in the Macintosh Inside Out Series (Addison-Wesley).

Throughout this chapter and the remainder of this book, you will see references to APDA, the Apple Programmer's and Developer's Association. This is an Apple group that distributes almost all of Apple's technical documentation. To order anything from APDA, you must sign an agreement and pay an annual membership fee, because you can buy beta versions of some development tools and documentation.

To join APDA, you can call 800/282–2732 in the USA, 800/637–0029 in Canada, or 408/562–3910 from anywhere else, or write to the following address.

APDA
Apple Computer, Inc.
20525 Mariani Ave., MS 33-G
Cupertino, CA 95014
USA

If you will never use beta versions of tools or documents, you can join an alternative program known as Developer Express. Call APDA for more information.

1 ▶ Overview of System 7

▶ Introduction

In this chapter, you will be introduced to the strategy behind and the history of System 7, then you will quickly look at all the components of System 7. The components are presented starting with those most visible to users and working toward those least visible to users, which mirrors the structure of the remainder of this book. Finally, you'll look at the requirements for running System 7.

▶ The Strategy behind System 7

The Macintosh operating system, with its tight integration and powerful human interface, has gradually evolved since its introduction in 1984. Despite some big improvements, such as the Hierarchical File System (released with the Macintosh Plus) and MultiFinder, it has been a process of evolution, not revolution.

Contrast the situation in the MS-DOS world with what Apple is doing on the Macintosh. MS-DOS users now have three different paths to choose from. The most conservative path is to simply continue with MS-DOS, which is continuing to evolve. The riskiest path is to switch to OS/2 (a new operating system) and Presentation Manager (the user interface that goes with it), which requires some users to buy new hardware and new software. A middle path is converting to Microsoft Windows, a user interface that resides on top of MS-DOS. Windows can run on some, but by no means all, MS-DOS machines. Many existing MS-DOS applications will run at the same time as Windows, so existing software doesn't

need to be scrapped. This is a difficult, confusing choice for both the user community and developer community.

Other companies, especially Microsoft, have seen the advantages of Apple's sophisticated graphical human interface. Windows and Presentation Manager on Intel-based microcomputers, and Motif and NeWS on workstations, have raised the standard of the average computer-human interface. These interfaces weren't a reaction just to the Macintosh interface, but Apple's success clearly demonstrated the importance of a high-quality, consistent, and compatible interface.

Although the Macintosh interface is still superior to Windows and so on in many people's eyes, much less of a gap exists between the Macintosh interface and Windows than between the Macintosh interface and MS-DOS. Apple is therefore being forced to innovate at a faster pace than before.

In the Macintosh world, existing operating-system capabilities are being enhanced and new capabilities are being added, such as the Communications Toolbox, the Data Access Manager, and high-level events. These are much more than simple extensions to the operating system. Apple is redefining what an operating system is. It is incorporating features into the operating system that were previously in the arena of application software.

Why is Apple doing this? Apple has never been a follower, but rather a leader. The company took an enormous risk with the Macintosh, and that gamble has clearly paid off. Apple did not make a PC clone when many analysts and industry pundits were advising that such a machine was crucial to Apple's success.

This redefinition of what an operating system is means application developers can now use high-level services by making a handful of system calls. Previously, for example, talking to a database might require dozens or hundreds of calls to accomplish a task. By using the Data Access Manager, developers can accomplish this same task using only a few calls. This means that users can expect to see communications capabilities in any application where it makes sense simply because it is so easy for developers to include that capability now. In addition, because these communications capabilities are part of the operating system and because a standard user interface has been defined for them, users will expect to see a standard set of dialogs whenever they use communications functions.

The net effect of these changes will be that Macintosh applications will tend to be more powerful, but still easier to use, than applications on other platforms. By raising the level of functionality of the Macintosh operating system, Apple gains a competitive advantage over other companies. By taking advantage of these new features in your applications, you can gain an advantage over your competitors.

▶ The History of System 7

System 7 arrived during a turbulent period in the computer business. To understand the feature set of System 7, let's look briefly at its history.

The System 7 that was announced at the Worldwide Developer's Conference in 1989 is different from the System 7 that started shipping in 1991. Many features were dropped between the initial announcement and the first alpha test version available to developers. (The alpha version was first delivered to developers at the 1990 Worldwide Developer's Conference, about a year after its announcement!)

Note ▶ The Worldwide Developer's Conference is open to members of the Apple Partners and Associates programs. These two programs enable commercial, in-house, and educational developers to get technical and marketing information from Apple.

The delivery of System 7 slipped more than once. System 7 required an enormous amount of code, some of which had to be compatible with past versions of the operating system, and some of which was written for the first time. As with any large-scale software project, unforeseen problems occurred and the delivery of System 7 was delayed.

The schedule for System 7 could have been much worse. Fortunately for Apple, System 7 was not a complete rewrite of the Macintosh operating system. In fact, System 7 extends the boundaries of the operating system by either enhancing existing capabilities or creating new capabilities. Few of the System 7 features depend on other new features. Certain features, however, such as high-level events, were crucial to the new operating system. A slip in the schedule for a critical feature meant that the schedule for the entire operating system slipped. Other components were removed from System 7 if they did not meet the System 7 schedule. These components will presumably be added to the operating system sometime after the initial release of System 7. Chapter 21 discusses possible futures for System 7.

▶ The Components of System 7

Apple has both improved existing portions of the operating system and added new components. The following sections give an overview of each part of System 7.

▶ The Finder

The Finder was completely rewritten for System 7. Although it is similar in appearance to the previous versions of the Finder, it has been remarkably improved. Users will notice changes to the Finder more than almost any other feature in System 7.

The Font/DA Mover is no more. To install fonts, desk accessories, or sounds, the user simply drags them into the System Folder. The System Folder has been simplified, and it now contains a set of folders that organize its contents along functional lines. For example, all preferences files go into the Preferences folder, and all control panel devices go into the Control Panels folder. When the user drags files into the System Folder, the Finder tries to place the file in one of these special folders.

For the first time, the Finder helps users find files. Using the Find command, you can select files using one criterion, such as file name, kind, creation date, and so on.

Users can put any desktop object in the Apple menu, including desk accessories, applications, folders, and documents. The user accomplishes this by moving those objects into the Apple Menu folder in the System Folder. The list of open applications that was located at the bottom of the Apple menu has moved to the right-hand side of the menu bar. They are now located in the Application menu. Desk accessories are launched into their own memory partition, so they also appear in this menu.

Apple has also made numerous other changes to the Finder, such as improvements in viewing choices, aliases, and File Sharing. The Finder interface has been substantially improved and made more consistent.

▶ The Human Interface

The Macintosh human interface guidelines have been revised to take System 7 into account. The Finder illustrates many of these new guidelines, which are described in *Inside Macintosh*, Volume VI.

For example, new types of icons are now supported by the Finder, including large icons (32 bits by 32 bits) and small icons (16 bits by 16 bits). Color icons are supported in addition to black-and-white icons. The guidelines suggest how to design the most effective icons in all these cases.

The new guidelines provide for movable modal dialogs. The user can move this type of dialog around in case it obscures something the user needs to see in order to respond to the dialog. The guidelines also support keyboard navigation in dialog boxes. Experienced users will appreciate this when, for example, they key through the scrollable list of files in the standard output file dialog.

Other guidelines cover the new features of the operating system, such as the Edition Manager. They include support for TrueType, the new font technology, which can produce smooth, nonjagged fonts at any (integral) size.

▶ The Installer

The Installer, the standard utility used to install the system, has been improved. Any application developer can write an Installer script for his or her users to install software, although it becomes especially useful when the software package involves placing files in special folders, copying resources, and so on. However, other installation programs are as useful or more useful than the Installer if the entire collection of files is simply copied to a single folder.

Version 3.1 of the Installer allows users to select a single button to install an application. The Installer script consists of a set of resources. Some of these resources describe rules that the Installer follows to decide which files or resources should be copied and so on. These resources, while nontrivial to create, make the installation of complex software packages relatively easy for users. A complex package, for example, might include an application, an INIT, and some resources that depend on which Macintosh the software is installed upon.

This Installer supports two important new capabilities. First, it supports "live" installs. That is, the user no longer needs to quit from other applications to run the Installer. Second, it supports installation over a network.

▶ Interapplication Communications and High-Level Events

System 7 greatly expands the idea of events. Previously, applications received low-level events, such as when a key was pressed down or when the mouse button was released. Applications can now receive high-level events, which you can use to provide high-level capabilities.

The most important class of high-level events is called Apple events. Apple events are standard sets of high-level events. The Finder uses a set of required Apple events to communicate with applications. These events are "open," "open document(s)," "print document(s)," and "quit." Other sets of standard Apple events cover generic handling of text and graphics. Yet other sets of them will be created to manage the capabilities of various types of applications, such as databases, spreadsheets, and word processors. You can also create proprietary high-level events. Only applications that were written with these events in mind will be able to use them.

It is very easy to use Apple events, interapplication communications

(also referred to as the PPC Toolbox), and high-level events over the network. In fact, from the programmer's point of view, there is little difference between using these services locally or over a network.

Apple events are in many ways the most significant change that System 7 brings. Although to users they are not the most obvious aspect of System 7, they portend a major change in the way we interact with computers. As more and more applications support more and more sets of Apple events, users will be able to connect their applications in interesting ways. Users will gradually stop interacting with applications and start interacting directly with documents. This is a much more natural way to use computers than what we do today.

▶ The Edition Manager

The Edition Manager, new with System 7, provides a new way of sharing data between applications. It is similar to the Clipboard in that a user must select object(s), but rather than copy to the Clipboard, the user "publishes" the selection. It becomes an edition file. Other users can then subscribe to the edition file and place the contents into documents, much like pasting from the Clipboard.

The difference between the Clipboard and the Edition Manager is significant. To illustrate this, let's look at an example. Let's say that the data being passed from one application to the other is a picture. Once you paste the picture into a text document, it remains the same. If the picture in the graphics program changes, you have to go to the graphics program, select the picture, copy it to the Clipboard, switch to the word processor, select the old picture, and paste from the Clipboard. If you have only a couple of pictures, that isn't much work. If you have dozens or hundreds of pictures, keeping the text document up to date becomes a tedious process.

On the other hand, if you use the Edition Manager to publish the picture, life is much easier. Simply subscribe to the edition and place it in your document. Whenever the picture has changed and you open your text document, you will see the latest version of the picture. The Edition Manager handles much of the magic here, but you will have to write some code to enable the Edition Manager.

▶ TrueType and Fonts

System 7 brings TrueType, Apple's new font technology. TrueType was born out of the technical limitations of bitmap fonts (Apple's previous font technology) and out of Apple's lack of control over PostScript. Before System 7, fonts were stored as bitmaps in fixed sizes. The Font Manager would do its best to provide a bitmap for a particular font in a particular

size in a particular style. In many cases, though, it had to resort to scaling the bitmaps from a different size. The results varied from passable to downright ugly.

TrueType fonts are stored as font outlines. When a font in a particular size is requested, the bitmap for it is generated on the fly. The results always look good because the resultant bitmap was algorithmically calculated for that size rather than scaled.

For the most part, application programs are not affected directly by TrueType. Restrictions on font sizes should be removed, however, since TrueType can provide fonts in any (integral) size.

TrueType means that high-quality output is available from lower-cost output devices. Specifically, it means that the output from a non-PostScript printer, such as a LaserWriter IISC, is of the same quality as from a PostScript printer, such as the LaserWriter IINT.

▶ TextEdit and International Services

TextEdit, the basic text-editing service, has been improved with System 7. This latest version has been completely internationalized, supporting the Script Manager. This means that in addition to supporting multiple fonts, sizes, and styles, TextEdit supports two-byte characters (used for Japanese, Chinese, and Korean) and mixed-direction text (left-to-right scripts and right-to-left scripts). Routines are also provided to calculate word and line breaks.

The Macintosh is an accepted machine not just in the United States, but in many other countries. The European countries and Japan are becoming big markets for the Macintosh. Over 40% of Apple's revenues are from sales outside the USA. The international services provided by the Macintosh operating system make it fairly easy to create software that can be internationalized. If you want to do this (and you should think about it seriously, because these markets are growing faster than the U.S. market), do it when you create your software. Although some kinds of software can be retrofitted, other kinds of software, especially text-intensive software, cannot be retrofitted without a great deal of work. By putting in a small amount of additional work in the beginning, you can take advantage of these other markets later on with little additional work.

▶ The Data Access Manager

The Data Access Manager and the Data Access Language (DAL, which was previously known as CL/1) are integrated into the system software starting with System 7. The Data Access Manager (DAM) provides a standard programmatic interface to relational database management systems.

The DAM is intended initially to provide access to relational database management systems that run on minicomputers (such as Ingres, Informix, and Oracle) and mainframes (such as DB2). The DAM, by presenting a generic interface to relational databases, enables users and applications to decide which database to work with at runtime, instead of being hard coded into the application.

You can use the Data Access Manager in one of two ways. The simplest, and least flexible, is to use a single system call. This call sends a query to a remote database and returns the results. The second method, which is a little more complicated but considerably more powerful, uses a dozen other calls to the Data Access Manager. These calls allow applications to modify queries on the fly and so on. In either case, the DAM establishes a connection with the remote computer, starts up the remote database management system, translates the query into the proprietary query language, retrieves the results, and converts them if need be.

Queries are written using the Database Access Language. This language provides a superset of the capabilities of SQL, the standard relational query language.

As the Data Access Manager evolves, it may become the standard programmatic interface to databases, both local and remote. If database vendors commit to the DAM by writing a low-level "database driver," known as a *ddev*, applications may be able to read from and write to most Macintosh databases as well.

▶ The Help Manager

The Help Manager, introduced with System 7, provides the first standard help facility in the Macintosh operating system—"balloon" help. When this feature is enabled, as the user moves the pointer over a screen object that has help messages available, the Help Manager displays the help message in a cartoon-like balloon. The content of the balloon tells the user what the object is and how it can be used. This is an interactive help facility and is not modal like many other help systems.

For the most part, you can provide help in most applications by adding special resources. In certain cases, such as when your application uses custom MDEFs (Menu DEFinition procedures), you'll have to write code to support the Help Manager.

The help available under System 7 will be useful primarily for novice Macintosh users and for anyone learning a new application. Other forms of help are required for other situations. Apple has clearly left room for additional help facilities in the future.

▶ The Sound Manager

The Sound Manager has been enhanced in System 7 to support some new capabilities. The Sound Manager now supports playing sounds continually from disk while application(s) are running, playing and mixing multiple channels of sound in real time, monitoring the CPU load that the sound channels in your application and others are making, and monitoring the status of sound channels.

An important new capability, released with the Macintosh LC and IIsi, supports sound input. An analog-to-digital converter built into the latest Macintosh computers can take an analog signal from a variety of sources and convert it to digital sound. The digital version of the sound can then be recorded to disk or played. The Sound Manager also supports external sound input hardware. This new capability will be used to implement voice mail and voice annotation of documents.

▶ The Communications Toolbox

The Communications Toolbox, introduced shortly before System 7, is now integrated into the system. This toolbox provides a standard programmatic interface to file transfer, terminal emulation, and connection protocol services. Users can easily switch from one connection protocol to another (and from one file transfer protocol and so on) by selecting another file transfer tool, another terminal emulation tool, or another connection protocol tool. These tools are external pieces of code, allowing for great flexibility in their use. Users also benefit from the Communications Toolbox because it provides a standard user interface for many aspects of communications. This reduces training time and, to a certain extent, the amount of documentation.

The Communications Toolbox consists of five managers: the Connection Manager, the Terminal Manager, the File Transfer Manager, the Communications Resource Manager, and, lastly, the Communications Toolbox Utilities. This general communications framework simplifies programming for communications services. Applications can now provide communications services whenever and wherever they make sense. You can provide these services with only a moderate amount of work.

▶ AppleTalk and File Sharing

The System 7 version of AppleTalk, the local area network protocol suite that is built into every Macintosh ever shipped, now includes Phase II support. AppleTalk Phase II was announced and shipped before System 7, and

it has been incorporated into the system. Many of the Phase II features do not affect applications at all, but some are especially useful.

The AppleTalk Data Stream Protocol (ADSP) is included as part of System 7. This protocol is easier to use for creating applications than the other protocols in the AppleTalk protocol suite. You can use ADSP to implement both clients and servers, as well as to implement peer-to-peer systems.

File Sharing brings a file server to everyone's machine. It provides a subset of the features (and performance) of the AppleShare software, which requires a dedicated machine. With Macintosh File Sharing, application programmers can assume that a file server is always available. Therefore, multiuser software will be easier to develop. When users need more performance from the file server, they can upgrade to AppleShare or to another AppleShare-compatible server.

▶ QuickDraw

System 7 incorporates 32-bit QuickDraw, the third major revision to QuickDraw. The first version supported black-and-white and "classic color," where eight colors (including black and white) are available. The first machine capable of displaying color was the Macintosh II, and simultaneous with its release came Color QuickDraw. This version supported 8-bit color, meaning that up to 256 colors can be used at a time. The third version of QuickDraw supports 32-bit color, which means that millions of colors can be used at a time, if your hardware supports it.

The latest version of QuickDraw also provides improved support of gray-scale displays, extensions to the PICT format to support 32-bit color and font names, and improvements to the **CopyBits** routine.

The Color Picker Package, which applications use to request a color from users, has also been improved. A new set of routines allows applications to convert colors from the RGB (Red-Green-Blue) model to the CMY (Cyan-Magenta-Yellow), HSV (Hue-Saturation-Value), and HLS (Hue-Lightness-Saturation) models.

The Palette Manager provides support for maintaining palettes of colors. The Graphics Device Manager provides an interface to graphics output devices. This manager is used primarily by low-level code, but not much by applications.

The Picture Utilities Package, introduced with System 7, provides routines that make it easy to examine PICT files and PixMaps. For example, you can use these routines to get the horizontal and vertical resolution, the number of lines or rectangles in the picture, and the fonts, sizes, and styles in the picture.

▶ The Memory Manager and Virtual Memory

The largest change in the Memory Manager is the arrival of virtual memory (VM), which is available only on machines with a memory management unit (Macintoshes with a 68020 and a 68851, or a 68030, or a 68040, support VM; all others do not). Virtual memory allows a disk file to substitute for additional memory. Obviously, virtual memory cannot be as fast as RAM, but the goal of VM is to permit users to buy RAM for their average usage. VM can be used during times of peak usage, such as when the user wants to run an additional application. Virtual memory does not affect application programmers very much at all. On the other hand, the performance of applications that are memory intensive, such as multimedia and scanner applications, may be adversely affected by VM. Some device drivers (especially disk drivers and NuBus drivers) will be affected by VM and may need to be rewritten to work with it.

Temporary memory, the heap memory not used by any processes, is much easier to use. Applications can use temporary memory when additional memory is required for short periods of time, such as opening or printing a document. Previously, the system software imposed severe limitations on how it could be used. You can use regular Memory Manager calls, for the most part, with temporary memory. Applications can now hold temporary memory for reasonably long periods, whereas before it had to be released before making calls to the Event Manager.

Applications should now be "32-bit clean." *32-bit cleanliness* means primarily that applications use 32-bit memory addresses. The older versions of the Macintosh operating system used 24-bit addresses, and some applications took advantage of this by using the extra byte. There are additional rules for 32-bit cleanliness. For example, CDEFs and WDEFs require a small change to be 32-bit clean.

▶ Processes

The Macintosh operating system currently supports a form of multitasking known as cooperative multitasking. This means that applications must follow certain rules if the system is to appear responsive to the user. Introduced with MultiFinder, this form of multitasking has now been completely integrated into the system. Under System 7, users can no longer run in Finder mode—that is, the system can always run more than one application. Although the Macintosh operating system does not yet support preemptive multitasking (as UNIX does), System 7 lays some of the groundwork for it.

The Process Manager, arriving with System 7, brings the concept of

process serial numbers (PSNs), which uniquely identify each running application or desk accessory. A set of routines allows software to get information about any process, bring a process to the front, launch a desk accessory, or launch an application.

The Notification Manager, introduced with MultiFinder, provides methods for software running in the background to notify the user. These services are one-way only—software can talk to the user, but the user cannot talk to software running in the background. (The user might have to bring an application to the foreground to communicate with the software. The Print Monitor is an example of this.) These services are an important addition to the user interface because the user would be quite surprised if the background software were to suddenly jump to the foreground. The Notification Manager can be used by applications, device drivers, and so on.

The Time Manager has been enhanced with System 7 to provide drift-free fixed-frequency scheduling services for tasks. Basically, the Time Manager maintains a queue of small tasks that it executes at the specified time. The latest version provides much more accurate timekeeping than previous versions. You can also use these services to make accurate measurements of elapsed time.

▶ The File System

The file system has been significantly enhanced in System 7 with the addition of file IDs, FSSpec records, the **PBCatSearch** system call, aliases, changes to the standard file package, Finder information, and the new structure of the System Folder.

File IDs provide a unique ID for each file on a volume. Previously, files on a volume could be tracked only by name. If the user renamed a file, there was no way to find the file using its old name. File IDs now enable users to find a file again not only if its name has changed, but also if its location on that volume has changed.

FSSpec records are a new way of specifying files when using File Manager calls. Instead of using various combinations of volume reference number, working directory ID, drive number, directory ID, partial pathname, and/or full pathname to specify a file, the new calls take an FSSpec record instead. One call creates an FSSpec record using various combinations. FSSpec records make it easier to save file location information.

The **PBCatSearch** routine provides a single call to search a volume for files meeting a set of criteria. The search criteria can be as simple as matching a file name, or they can involve much more complex parameters

such as file size, file creation date, file type, and so on. (Previously, this function would have required several pages of code to accomplish.)

Aliases, which effectively provide more than one location for a file, are a new capability. An alias behaves, for the most part, just like the file to which it points. You can use an alias to provide a generic file reference so that even when the file it points to is changed, the alias does not have to change.

The Standard File Package, enhanced with System 7, is used by the majority of Macintosh applications when users open or save files. The latest version of this package has an improved user interface, making users happy; it is also much easier to customize than before, making programmers happy.

Finder information is now available to applications that need it through a new set of calls. This information includes application signatures (that is, the application creator type and the file types it owns), icons for applications and documents, and the user's comments. This information was previously kept in an invisible file called Desktop, and it was not accessible to applications.

The structure of the System Folder has changed for the better. Previously, the System Folder held many different kinds of files. It was not uncommon for a System Folder to contain more than a hundred files. The lack of organization in this folder made maintenance a headache. The new architecture of the folder means that files are organized into functional categories, such as preferences files, control panel files, and items (such as applications) that should be opened at system startup time. These special folders are accessible to applications by a new system call.

▶ The Hardware Managers

The Slot Manager, which provides a programmatic interface to NuBus cards, has been enhanced in System 7 to work with 32-bit addresses. The routines provided by this manager are used primarily in device drivers for NuBus cards. Application programmers will rarely need to use this manager.

The Power Manager, introduced with the Macintosh Portable, provides applications, drivers, and system software with control and access to the electrical power of the Macintosh Portable. The Portable is never off; if it isn't active, then it is either in the idle state (when the CPU runs at a slow clock speed) or asleep (when the CPU is not running, but some circuitry is active, waiting for the user to restart the machine). Applications can be left open in both the idle and sleep states. Some applications may want to control when the machine enters either of these states.

▶ Running System 7

This latest version of the Macintosh operating system provides many new capabilities. Because so much has been added to the operating system, new requirements exist for the hardware required to run it: requirements for RAM and requirements for running virtual memory.

▶ Memory Requirements

The requirements for running System 7 are simple: You need a minimum of 2 megabytes (Mb) of random access memory (RAM) to run System 7. Every Macintosh, from the Plus on, can run System 7.

Apple could reduce the memory requirements of future machines by moving portions of System 7 into the ROMs of new machines. It isn't clear that this will be worth doing, though. The semiconductor industry is shipping 4 megabit (4Mbit) RAM chips today and is developing 16Mbit chips in the next couple of years. Five years ago, 64K of RAM was a lot of memory, but today it's not much at all compared with the amount of memory in a Macintosh Plus. In the same way, the members of the Macintosh family arriving in two years will probably have lots of RAM compared to the Macintoshes of today.

▶ Virtual Memory Requirements

Virtual memory provides a less expensive source of memory than additional RAM chips. To run virtual memory (VM) under System 7 (and any future versions), you must have a memory management unit (MMU). This is a chip that helps the CPU translate memory references from a logical address space into physical memory. This translation process has to take place in hardware, because it must happen quickly. It is not possible to do this in software and have an acceptable level of performance.

No MMU chip can work with a 68000 CPU chip, so any machine based on a 68000 cannot run VM. However, such machines can run most other portions of System 7 (except Color QuickDraw and related managers).

Macintosh computers with a 68020 CPU chip can run VM if you install a Motorola 68851 PMMU chip. The PMMU chip is a memory management chip designed to support virtual memory for the 68020 CPU. The AMU chip that was shipped in the original Macintosh II is *not* an MMU. This chip performs a simple address translation and cannot perform the translation needed to support VM. You must replace it with the Motorola chip if you want to run virtual memory.

Macintoshes based on a 68030 or 68040 chip have an MMU built into the CPU chip. All such machines can run VM.

Apple recommends that you have no more than twice as much virtual memory as physical memory. You may use more or less virtual memory, but the operating system has been optimized for this ratio.

Another requirement for running VM is disk space. Virtual memory produces a larger address space by keeping active *memory pages* (a memory page is a contiguous chunk of memory) in memory. Idle memory pages (a chunk of memory) are stored in a disk file. This file must be as large as the total amount of virtual memory, not the difference between virtual and physical memory. This is because VM needs to be able to move *any* page of memory to disk. For the sake of efficiency—and VM must be very efficient—the mapping between virtual memory and this disk file must be simple. It's easiest to use a disk file as large as the size of virtual memory, because the mapping between the location of a page in virtual memory and its location in the disk file is trivial. Figure 1-1 illustrates this simple mapping.

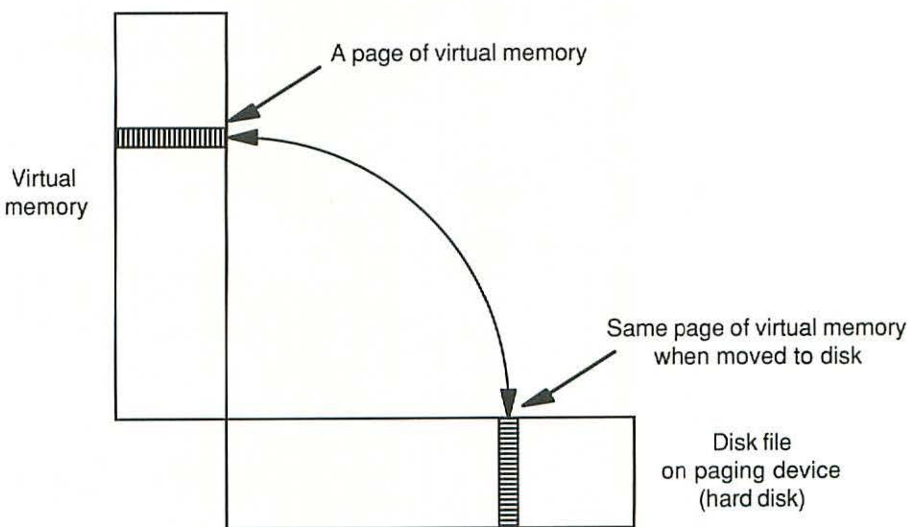


Figure 1-1. The mapping between virtual memory and disk

▶ Conclusion

In this chapter, you first learned about the strategy behind System 7 and its history. With System 7, Apple has redefined what an operating system is by increasing the level of functionality. You have also been introduced to each of the major components and the requirements for running System 7.

Get Info ▶ For more information about each of the System 7 components, refer to the succeeding chapters in this book. At the end of each chapter, you will find references to additional technical information.

2 ▶ The Finder

▶ Introduction

The Finder is a component of the standard system software that provides file and process management capabilities. The Finder distributed with System 7 has been completely rewritten in C++. Its interface is compatible with older versions of the Finder, but has nonetheless been significantly improved. In this chapter, you'll look at the major changes to the Finder.

The interface has a few big changes and many small changes. The Finder is the one place where every user will feel the difference between System 7 and earlier systems. Apple has clearly spent a lot of time polishing the interface. The new Finder answers most of the complaints about the earlier Finders. It also provides an excellent example of how a user interface can evolve and yet remain compatible with older versions.

▶ The Menu

At first glance, the menu bar on the new Finder doesn't look that different from earlier Finders. It has a new menu named Label, but that seems to be about it. Look a little closer, and you'll see that the menu bar actually has quite a few changes. Let's start with the Apple menu and work toward the other side, from the user's point of view. The Finder's menu bar is illustrated in Figure 2-1.

 File Edit View Label Special

Figure 2-1. The Finder's menu bar

▶ The Apple Menu

The Apple menu has changed. No longer does it contain a list of desk accessories and a list of open applications. It now contains a list of desktop objects. Users can add anything on the desktop to the Apple menu, and when they select that object from the menu, it is opened by the Finder. You can still have desk accessories in the Apple menu, but you can also have applications, a document file, or a folder. How do you add things to the Apple menu? By moving them into the Apple Menu folder, which is in the System Folder. Where did the list of active applications go? See the Application menu.

▶ The File Menu

The File menu now has a "Find..." command. For the first time, the Finder finds files. Previously, you would have used Apple's Find File desk accessory (DA) or a program like On Technology's On Location to track files down. The "Find..." command provides much stronger search capabilities than the Find File DA, but it doesn't provide all the capabilities of On Location, which can search files by content as well as name.

By choosing the "Find..." command, you'll be presented with either a simple dialog or a more complex dialog. The two versions of this dialog are shown in Figure 2-2. When you enter a search criteria and then press the Find button, the Finder searches for one or more files that meet your criteria. If a file is found, then its folder is brought to the front and its icon is highlighted. To find the next file meeting your criteria, choose the "Find Again" command.

▶ The Edit Menu

The Edit menu under the Finder has not changed. However, the Edit menu under many applications has changed with the addition of commands related to Publish and Subscribe. See Chapter 7 for more information.

Find
Find: resedit
More Choices Cancel Find

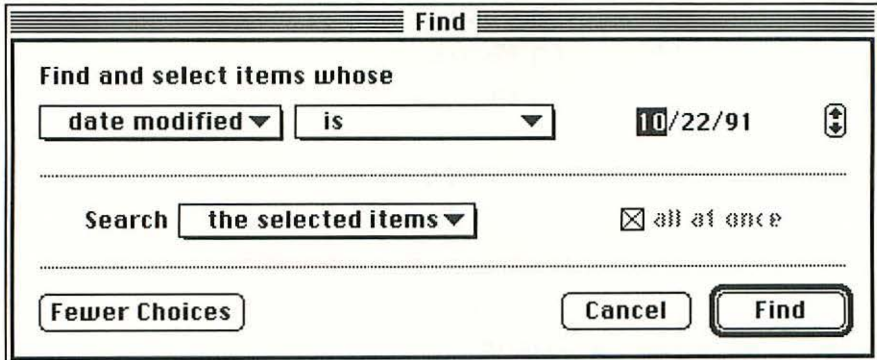


Figure 2-2. The Find dialog box

▶ The View Menu

The View menu has not changed a lot from previous versions, except that you can view by label in addition to the previous choices. This is explained shortly. When you are viewing files in a list, though, you'll see small triangles next to folders. By double-clicking on one of these dots, the contents of the folder will be expanded and displayed, or collapsed. This allows you to view a hierarchy of files without having to open and close folders. A list of files is shown in Figure 2-3.

▶ The Label Menu

The Label menu is new with System 7. In addition to the default label None, you can create up to seven labels for your files. By selecting one or more objects on the desktop, you can then use this menu to label them. Why would you want to do this? Perhaps you have two major responsibilities. All the files associated with your first job could be labeled differently from those associated with your second job. The View menu now supports viewing by label, so labeling files can be a useful feature. Files of

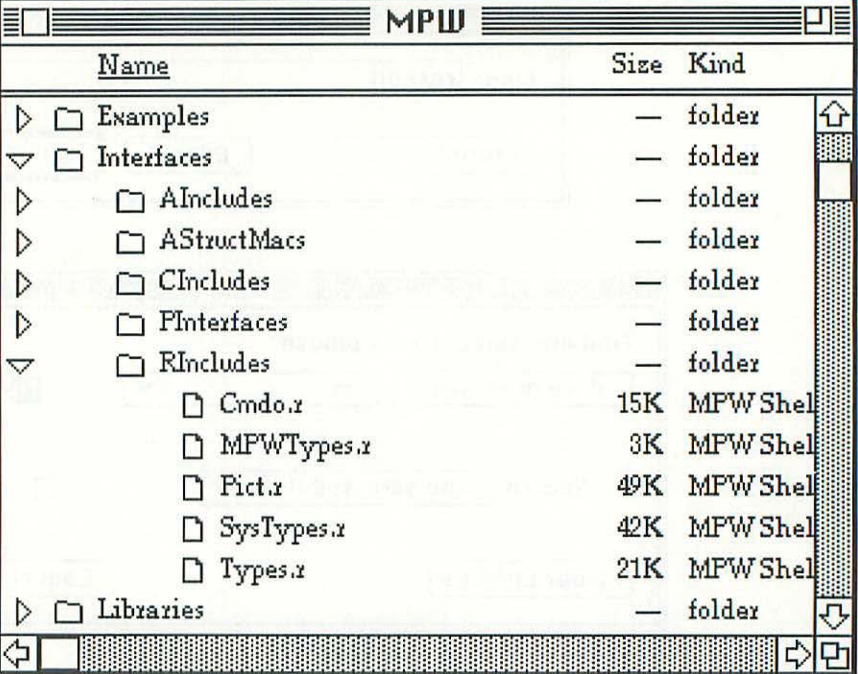


Figure 2-3. A list of files

each kind of label are displayed in a different color, so here is one place when a color machine is an advantage.

▶ The Special Menu

The Special menu has changed in a subtle way. If you look at the "Set Startup..." command, you'll see that you can no longer choose between Finder and MultiFinder. With System 7, MultiFinder has been completely integrated into the system. You can no longer run under the "uni-Finder."

▶ Application Menu

That's about it for the left-hand side of the menu bar. Let's see what the right-hand side looks like. You'll see either two or three icons on the right. The rightmost icon is the Application menu. On older systems, when you clicked on this icon, the current application was moved to the background and the next application was brought to the foreground. Now this icon

behaves like a menu. Its primary purpose is to display a list of active processes. That means active desk accessories as well as active applications are listed separately. You can also choose to hide the windows of the active application and to show only the active application.

▶ Script Menu

The next icon depends on whether you are using any script systems besides the Roman script system (which exists in every system). If you have installed any other script system, such as Japanese or Hebrew, you will see an icon for the current script system on the menu bar. This icon is the title of a menu that allows you to switch between any of the script systems that are currently installed on your system. If the software you are using has been designed for using multiple languages, you can use several different languages and scripts in a single document.

▶ Help Menu

The last icon (the leftmost icon on the right-hand side of the menu bar) is the Help menu. This menu allows you to turn balloon help on or off. If balloon help is on, then the Help Manager displays a help balloon whenever the pointer is positioned over an object that has help messages. The purpose of balloon help is primarily to answer the question, "What is this object and what does it do?" While balloon help is on, the mouse behaves as always. In other words, even though help balloons are being displayed, clicking, dragging, double-clicking, and so on behave normally.

▶ The System Folder

System 7 brings major changes to the System Folder. Before System 7, System Folders tended to accumulate large numbers of files, the functions of which were rarely obvious. Let's look at how the new System Folder creates more order, and then say goodbye to the Font/DA Mover.

▶ The New System Folder

The System Folder under System 7 has a new look. Rather than dozens or hundreds of files accumulating in one folder, the many files that belong in the System Folder are now grouped by function into a set of special folders inside the System Folder. These folders are listed in Table 2-1, and the System Folder is shown in Figure 2-4.

Table 2-1. The special folders in the System Folder

Folder Name	*Contents*
Apple Menu Items	Items that should appear in the Apple menu
Communications Folder	Correction, File Transfer, and Emulation tools for use with the Communications Toolbox
Control Panels	Control panel files (for organization only—control panels can be opened anywhere)
Desktop Folder	Information about the icons appearing on the desktop; this file is invisible.
Extensions	Software that provides system-wide functionality, including INITs and printer drivers
Preferences	Preferences files created by any application
PrintMonitor Documents	Temporary files created when printing
Rescued Items from *volume name*	Items that were located in the Temporary Items folder the previous time the system was booted. Since applications normally remove items from the Temporary Items folder, the existence of this folder indicates that a system crash occurred. The user can choose to delete these temporary files or try to recover their data.
Startup Items	Items that should be opened every time the system starts
System	Note that this file also contains the basic system software, but behaves as a folder with respect to fonts, sounds, and other moveable resources.
Temporary Items	Temporary files created by applications; this file is invisible to users
Trash	Items that the user has moved to the trash icon.

When you copy a file to the System Folder, the Finder attempts to figure out to which of these special folders (if any) the file belongs. Before moving the file into a special folder, the Finder will ask you to confirm its choice.

▶ Font/DA Mover

With the new System Folder architecture, the Font/DA Mover has been retired. This application, which was used to install fonts and desk accessories into the System file, had a complex, cumbersome interface that was also difficult for many users to understand. Fonts are now installed by moving the suitcase file into the System file in the System Folder. The

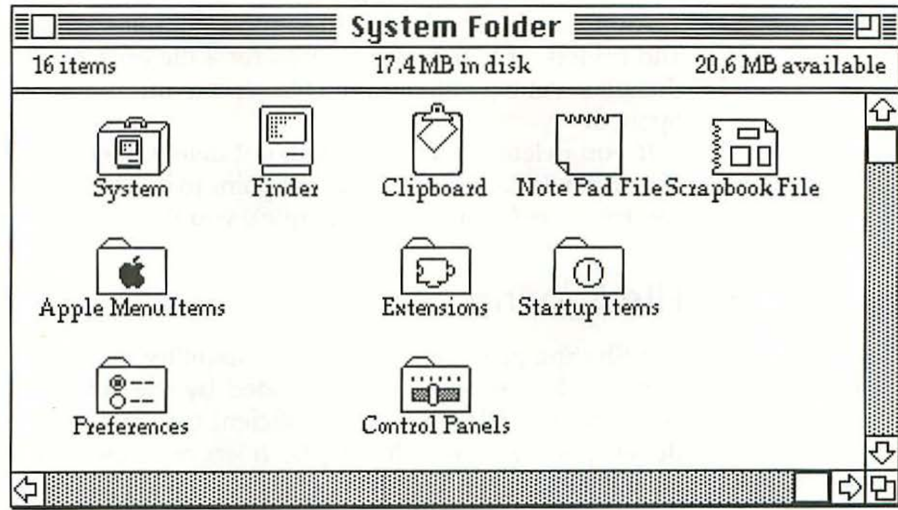


Figure 2-4. The new look of the System Folder

fonts are immediately available. Double-clicking on a font icon brings a window displaying information about the font. Sounds are installed in a similar manner. Double-clicking on a sound icon plays the sound.

Desk accessories can be located anywhere on a disk. You can open a desk accessory by double-clicking on it, just as you would an application. DAs that are in the Apple Menu folder appear in the Apple menu, but there is no longer any reason why they cannot be kept in other locations.

Clearly, desk accessories have not disappeared with System 7. There is no longer much reason to write something as a desk accessory though. DAs are harder to write than applications, primarily because they impose more restrictions on what you can do. Small applications make even more sense now. DAs will gradually fade away, but only because better methods of accomplishing the same thing are now available.

▶ Aliases

Aliases allow you to create a small file that "points" to the original file. You use aliases to put a file (or any other desktop object) in more than one place. For example, you might create an alias of an application so that you can put the alias in the Apple Menu folder. The original application can remain in a folder elsewhere, along with all of its associated files. The Finder displays aliases in italics so that they can be easily distinguished from all other files. When you open the alias, the Finder actually opens the original, no matter where it is located.

Another use for aliases is to simplify complex arrangements of volumes and folders. If you create an alias for a file on a file server, then opening the alias will log you onto the file server, and then the original file will be opened.

If you delete an alias, you will not delete the original file. If you delete the original file, the alias will still point to it. The next time you attempt to use the alias (open it, for example), you'll get a warning message.

▶ File Sharing

File Sharing provides a file server capability in any Macintosh. The services are the same as those provided by the full AppleShare file server software, but File Sharing is sufficient for small groups (less than a half dozen people). File Sharing is, however, lower in performance than AppleShare.

You can control whether File Sharing is on or off using the Sharing Setup control panel, as shown in Figure 2-5.

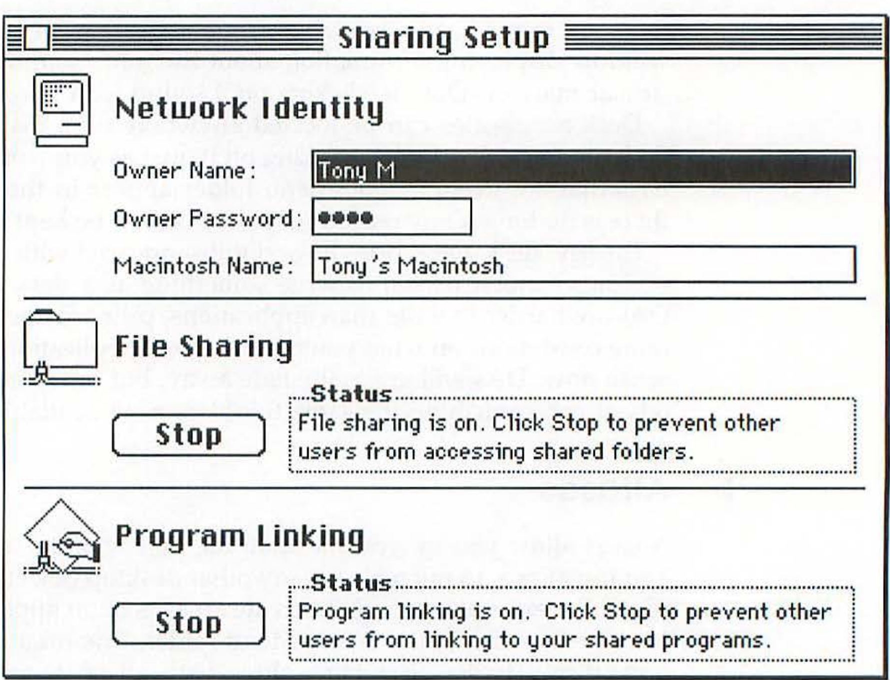


Figure 2-5. The Sharing Setup control

When you've turned File Sharing on, you can let others use your volumes or specific folders on volumes. Do this by selecting the object and then using the "Sharing" command from the File menu. You are then presented with a dialog that lets you specify who can access it and what kind of access they can have (read only, read and write, and so on). This dialog box is shown in Figure 2-6. To give access to a specific person or group of people, you can create a new entry in the Users and Groups control panel. This dialog box is shown in Figure 2-7.

Figure 2-6. Sharing a folder

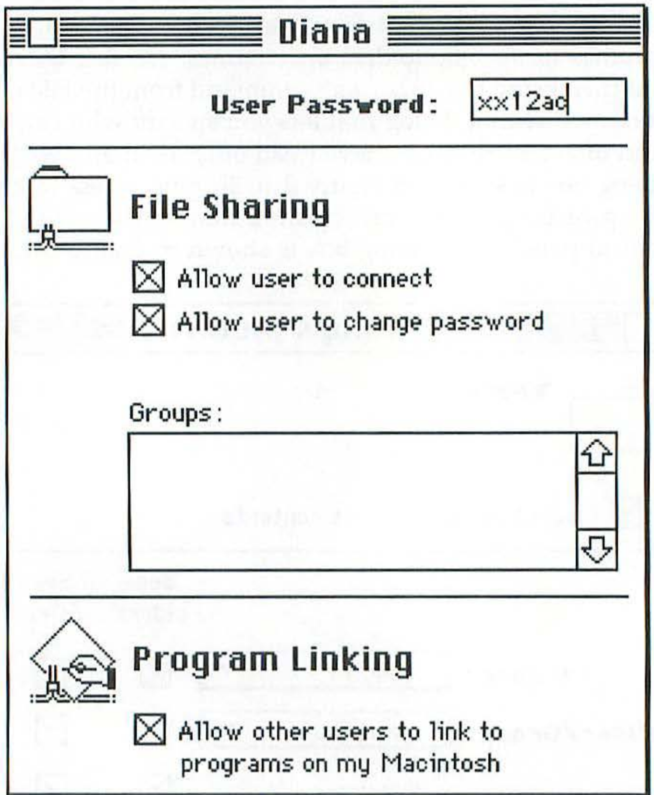


Figure 2-7. Users and Groups control panel

▶ Desktop Objects

Quite a few changes affect the objects on the desktop. Let's look at some of them now: navigating the desktop, stationery pads, and the Trash.

▶ Navigating the Desktop

In previous Finders, you could only navigate the desktop using the mouse. Now you can also use your keyboard. The arrow keys move the selection to the left, right, up, or down from the current selection. The Command-up arrow opens the parent folder, and the Command-down arrow opens the selected folder. The Command-Option-up arrow opens the parent folder and closes the current folder. The Command-Option-down arrow opens the selected folder and closes the current folder.

The Tab key moves to the next icon alphabetically. The Return key opens the selected icon's name for editing, and the Enter key closes it. Any other keys entered select the icon whose name begins with those letters.

▶ Stationery Pads

A stationery pad is a special document. You can make any document into a stationery pad by checking the stationery pad box in the document's Get Info dialog box. When you open a stationery pad, you will get an untitled window whose contents are the same as the stationery pad's. You can use stationery pads to simulate forms using most applications.

Applications need a small amount of code to support stationery. If this isn't there, the Finder simulates it by duplicating the stationery pad and opening the untitled copy.

▶ The Trash

The Trash behaves differently in several respects than it did before. For instance, the Trash is emptied only when you explicitly empty it. The system doesn't empty the Trash automatically when you start an application or shut the computer down. You can choose whether the Finder should warn you when you empty the Trash by selecting the Trash and using the "Get Info..." command.

The Trash now appears as a folder on the desktop when you're using the standard file dialogs. This is more consistent behavior than the previous implementation.

▶ Virtual Memory

Virtual memory (VM) substitutes disk space for (RAM) memory. The process of simulating memory with disk space must happen quickly. In practice, this means a lot must happen in hardware, and only some Macintoshes have the required hardware. Macintoshes that have a memory management unit can use this feature. Basically, a Macintosh with a 68020 CPU chip and a 68851 PMMU chip or with a 68030 or 68040 CPU chip can run virtual memory. Machines with a 68000 CPU chip or a 68020 CPU chip without a 68851 PMMU chip cannot run virtual memory.

If you can run VM on your machine, you need only buy sufficient RAM for your average needs. If you need to run one more application beyond what your RAM will support, you can run it using VM. Your system will not run as fast as it would if you had more RAM, but in many cases you may not notice the difference in performance.

The Memory control panel, shown in Figure 2-8, lets you control

whether VM is on or off. You'll need to restart your machine for this change to be made. You can also control how much VM you use. You'll need as much free space on a volume as the total amount of memory you want (not the amount of virtual memory).

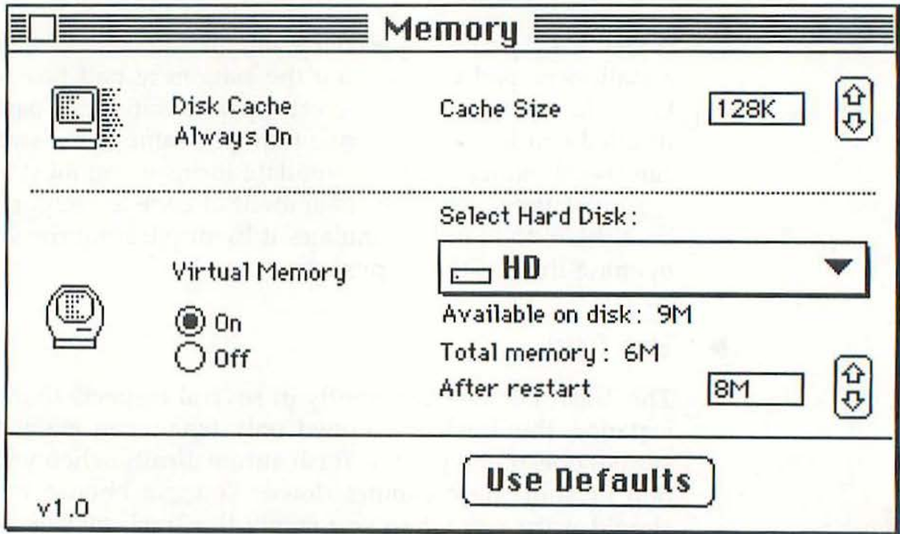


Figure 2-8. The Memory control panel

▶ Conclusion

In this chapter, you've looked at the new Finder. This Finder, which was completely rewritten for System 7, looks similar to previous versions of the Finder while offering numerous improvements in its interface.

The changes listed in this chapter are not all the changes in the new Finder. Apple has rethought the Finder interface quite thoroughly, so you will find some of them only after you've been using it for a while. Unquestionably, the Finder is a lot easier to use.

Get Info ▶

For more information about the changes to the human interface guidelines, refer to the user documentation distributed as part of System 7. The Finder is not documented in *Inside Macintosh*, Volume VI, except for certain programmer-level concepts.

3 ▶ The Human Interface Guidelines

▶ Introduction

The success of the Macintosh is due in part to the human interface guidelines. These guidelines, which were introduced with the first Macintosh, were quickly supported by the user community and the trade press. Any arbitrary deviations from the guidelines were punished by both users (who tended to buy other products) and the press (who gave lower ratings for nonconforming software).

In this chapter, you will look at the changes that System 7 brings to the Macintosh human interface guidelines. Actually, some of these changes are not new with System 7—programmers have been informally following them for some time, but only now have these changes been documented. The changes fall into four categories: color, menus, windows, and dialog boxes. You will learn about each of them in turn.

Another important aspect of the human interface guidelines is support for other languages. This aspect of interface design is discussed in more detail in Chapter 9, which covers text and international services.

▶ Color and Interface Design

The interface for System 7 uses much more color than any previous version of the operating system. Apple has used color to help users work more effectively, not simply to draw attention to the interface.

In System 7, you can use color in the following ways:

- Colored scroll bars, close box, zoom box, size box, and thumb and arrow buttons in windows and dialogs
- Color window frames that become a shade of gray when in the background
- Gray racing stripes and scroll bars

Users can change some of these colors using the new Color control panel.

You'll first look at some general guidelines for using color. Following the general guidelines, you'll look at the new kinds of color icons. Then you'll look at several new types of icons that you may choose to support in your applications.

▶ Guidelines for Using Color in the Interface

Your applications should use color in the same way as the system software—to aid the user, not to distract him or her. Keep in mind that color should not be used to convey important information, because not all Macintoshes can display colors and because quite a few people are color blind.

The technique for designing with color is to design first with black and white. After you have a successful black-and-white design, then add color. It is more difficult to design first in color and then convert the design to black and white. Apple recommends that you use as few colors as possible and preferably to use a 4-bit palette, since it can be used on more machines than an 8-bit palette. Try to use light or subtle colors rather than bright colors in most cases. This will reduce the clutter on the screen.

The apparent light source on the Macintosh screen is located beyond the upper left-hand corner of the screen. Keep shading consistent with this.

▶ Icons

The Finder now supports several new kinds of icons. Your application can provide not only the standard 32-pixel by 32-pixel black-and-white icon, but also the following types of icons:

- 16-pixel by 16-pixel black-and-white icon
- 32-pixel by 32-pixel 8-bit color icon
- 16-pixel by 16-pixel 8-bit color icon

- 32-pixel by 32-pixel 4-bit color icon
- 16-pixel by 16-pixel 4-bit color icon

If no small icon is provided, the Finder uses an algorithm to create a small black-and-white icon as before. If you provide your own small icon, you can create a better-looking icon.

The Finder uses the appropriate type of icon. Whether it uses the black-and-white icon, the 4-bit color icon, or the 8-bit color icon depends on the current settings for the monitor on which the icon(s) will be displayed. Whether it uses the large or small icon depends on the current context and user options. Note that only one mask is used for each size of icon; that is, the same mask is used for all three 32-pixel by 32-pixel icons. A smaller mask is used for all 16-pixel by 16-pixel icons. The mask is used in conjunction with icons by the Finder to indicate when a file is selected, open, or both selected and open.

When designing your icons, design the black-and-white icons first; then colorize the icons. The color icons should resemble the black-and-white icons, or your users will get confused.

▶ New Kinds of Icons

The changes in System 7 also affect the kinds of icons that your application uses. In addition to icons for documents, you may wish to provide new icons for the following kinds of files:

- Stationery—A stationery file behaves like a regular document file, except that when opened, it is transformed into an untitled document
- Edition file—For use when your application publishes an edition file
- Query document—For storing a Data Access Manager query as a document

The Finder has default icons for all three types of files shown in Figure 3-1. These default icons are used if your application does not provide its own icons.

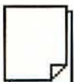


Figure 3-1. Default icons for stationery, edition, and query files

Menus and Interface Design

System 7 brings several new managers to the operating system. Some of these managers directly involve users in their activities, and so provide several new standard menu commands. You should add these commands to your application's menu bar only when appropriate. Let's first look at the new standard menus and then at pop-up menus.

New Standard Menu Commands

If your application uses the Data Access Manager, which is described in Chapter 10, you should add an "Open Query" command to the File menu. This command opens and executes a query document.

If your application uses the Edition Manager, which is described in Chapter 7, you should add a set of commands to the Edit menu. These commands should follow the cut/copy/paste set of commands and be separated from them by a gray line. The new commands for supporting the Edition Manager are "Create Publisher...," "Subscribe To...," "Publisher/Subscriber Options...," "Show/Hide Borders," and (optionally) "Stop All Editions...".

TrueType, which was introduced with System 7, provides bitmap fonts in arbitrary sizes. These bitmaps are generated from an algorithm and can therefore be provided in arbitrary sizes. TrueType is described in Chapter 8. Therefore, because the limitations of previous types of fonts no longer hold in all cases now, any restrictions that your application places on font sizes should be eliminated, or at least relaxed. Users should be allowed to specify an arbitrary (integral) font size. Fonts can now be larger than 127 points in size, so you can eliminate the size limitation when appropriate.

If your application provides help beyond the Help Manager, move your application's Help command to the Help menu. Users can then find all help under a single menu.

Pop-up Menus

The appearance of pop-up menus has been improved with the addition of a downward-pointing arrow. This arrow clearly distinguishes pop-up menus from editable-text fields. Figure 3-2 illustrates a pop-up menu with this arrow.

In some cases, pop-up menus should support user-specified values. With such a pop-up menu, the user can either select a standard value or type in another value. The nonstandard value should be displayed as the first item in the pop-up menu, and it should be separated from the standard values by a gray line.

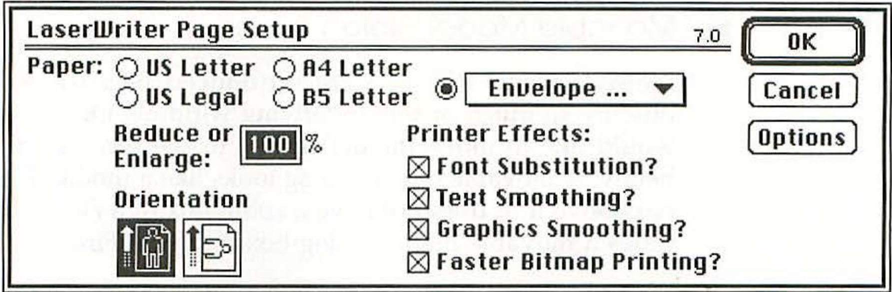


Figure 3-2. System 7 version of the pop-up menu

▶ Windows and Interface Design

Handling windows is more complex now because many different monitor sizes are available and because some Macintosh systems support the concurrent use of multiple monitors. System 7 brings revised guidelines for handling windows.

If reasonable, save and restore the position, size, and state of document windows. When users reopen a window, though, you may need to adjust these values to ensure that the window will fit onto the current monitor. For example, a user might create a document using a large monitor at work, then save the document and bring it home. When she opens the document on her Macintosh Classic at home, she should expect to see the document on the screen, regardless of where the document was positioned or sized at work.

You need to interpret the zoom box carefully because of the variety of monitor sizes. Remember, the zoom box toggles the window size between the window state the user has created and a "full" or standard state. When zooming to the standard state, use the screen wisely. Displaying a word processing document by using the entire screen of a large monitor is not useful. Documents have a natural width, and they should be displayed in a window appropriate to that width.

If the user has multiple monitors and the user opens more windows, the added windows should be displayed on the current monitor. This is not necessarily the monitor containing the menu bar.

▶ Dialog Boxes and Interface Design

System 7 brings two new features to dialog boxes: movable modal dialogs and keyboard navigation. These two changes make dialog boxes more usable. The other changes with respect to dialog boxes are changes to the human interface guidelines.

▶ Movable Modal Dialog Boxes

Movable modal dialogs were introduced because modal dialogs often obscure so much of the underlying window. On many occasions, a user would like to move the dialog box to see some portion of the window below. A movable modal dialog looks like a modal dialog box with a title bar above it. It does not have a zoom box or a close box. Figure 3-3 illustrates a movable modal dialog box from the Finder.

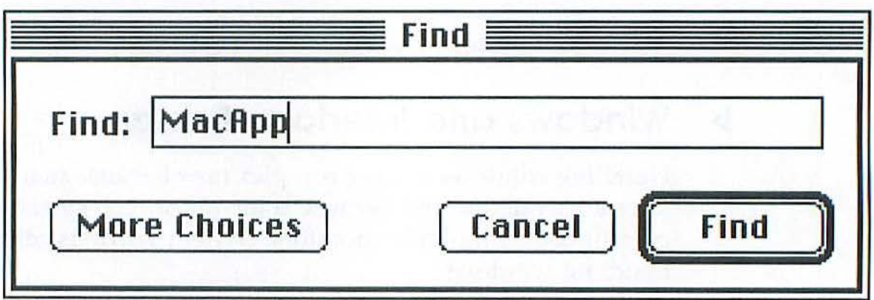


Figure 3-3. A movable modal dialog box from the Finder

By the Way ▶

A *modal dialog* is a dialog that the user must dismiss before anything else can be done. All mouse and keyboard events are processed by the dialog box until it is dismissed. This means, for example, that you cannot click on another window to bring it to the front. A *modeless dialog* behaves like a window in that it can be moved and will be sent into the background if the user clicks on another window.

These new dialog boxes behave like nonmovable modal dialog boxes, except that the user can drag the dialog box around using the title bar. Apple, however, recommends several changes. When using a movable modal dialog, you can allow your application to switch to the background. Obviously, you won't allow the user to bring another window from your application to the front—that's the purpose of a modal dialog box.

Apple further recommends that movable modal dialogs allow the user to access the Edit and Help menus. By allowing use of the Edit menu, the user can cut, copy, and paste into text fields. By allowing use of the Help menu, the user can more easily get help.

▶ Keyboard Navigation in Dialog Boxes

For the most past, *keyboard navigation* in dialogs has allowed access only to text-editing fields until now. On occasion, you'd like to use the keyboard in other places in dialogs, especially for scrolling lists. The standard file dialog box displayed when the user selects the "Open..." command allows keyboard navigation. By typing one or more letters, the user scrolls the list of files to the file that begins with those letters. However, the standard file dialog box displayed when the user selects the "Save As..." command has not allowed such keyboard navigation.

With the advent of System 7, keyboard navigation into scrollable lists is now encouraged by the guidelines. The latest version of the Standard File Package and the Chooser provide two examples of this feature. You should indicate that a scrollable list is the current item by drawing a rectangular border around it. (If an editable-text field is the current item, the blinking cursor provides feedback to the user.)

Speaking of the standard file dialog, System 7 brings many changes to it as well. The button previously named Drive is now named Desktop. By pressing this button, the user is presented with a list of all volumes and all files on the desktop. The Trash can appears as a folder on the desktop in these dialogs, just like any other folder. A new button, titled New Folder, allows users to first create a folder and then save the document. The programmer-level details of the new Standard File Package are described in Chapter 18.

▶ Changes to the Macintosh Human Interface Guidelines

Ideally, you should label buttons with a verb describing the action that will be performed should the button be pressed. Map the Return and Enter keys to the default button, which is the button that provides the most likely response or the safest result. Note that the most likely response is not necessarily the same as the safest result.

Whenever appropriate, provide a Cancel button. The Escape key and Command-period should be mapped to this key. Cancel should mean "return to the state prior to this dialog box." If the user presses the Cancel button, no side effects should occur. Therefore, if you are considering a Cancel button, but it won't have this meaning, name the button differently. Stop, Revert, and OK are examples of other labels that might prove useful.

Modal dialogs should provide feedback so that the user can see the changes. You can do this either by updating the document or by providing

an example in the dialog itself. Apple recommends that you do not use an Apply button, because it combines the meanings of both OK and Cancel.

Simple dialog boxes, such as alerts, should be consistent with the recommended spacing of elements as described in the User Interface Guidelines chapter of *Inside Macintosh*, Volume VI. The Action button should be located in the lower right-hand corner, with the Cancel button to its left.

In dialog boxes for saving all changes, Apple recommends that the most dangerous button be placed on the left-hand side of the box, aligned with the message text. In this way, the user has to move the mouse more to reach that button than the safer options.

▶ Conclusion

In this chapter, you've looked at the improvements to the human interface guidelines that System 7 has brought. The changes come in four categories: color, menus, windows, and dialog boxes. Apple has put a lot of thought and effort into improving the guidelines, as the new Finder clearly demonstrates.

Get Info ▶

For more information about the changes to the human interface guidelines, refer to the User Interface Guidelines chapter in *Inside Macintosh*, Volume VI. This chapter includes color illustrations, which are helpful in understanding the guidelines for using color. For information about the basic Macintosh human interface guidelines, refer to *Human Interface Guidelines: The Apple Desktop Interface* (Addison-Wesley, 1987). This book was written by the Human Interface Group at Apple, and it should be read by every Macintosh programmer. If you're involved with developing Macintosh software, you should have a copy of it on your bookshelf.

If you are writing user documentation, you should also have a copy of the *Apple Publications Style Guide*. This document, which is available from APDA, contains all the standard Macintosh terminology that you should use in documentation. Just as the Macintosh interface has a set of guidelines, this book documents the terms for all Apple user documentation. (Most other companies' Macintosh documentation also uses these terms.) If you use a common set of terms in documentation and avoid technical jargon, users will have less trouble reading documentation because they will already be familiar with many of the terms.

4 ▶ The Installer

▶ Introduction

In this chapter, you'll look at the latest version of the Installer, a standard system utility used by Apple to install new versions of the operating system, as well as code and resources for printing and networking. Some application developers also use the Installer, especially for software that cannot be trivially installed by copying files. Version 3.2 of the Installer ships with System 7.

The latest version of the Installer offers two levels of usage: a one-button mode, in which the Installer figures out what to install by using rules stored in the Installer script, and an expert mode, where the user can manually select what software will be installed. You'll first look at how to write Installer scripts. Then you'll look at how to customize the Installer by adding your own splash screen and by writing external code modules for the Installer application. The dialog for the one-button install option for System 7 is shown in Figure 4-1. The dialog for the complex install option is shown in Figure 4-2.

The new Installer provides two new important capabilities. First, it can install to the active system. This process, known as a "live" install, simplifies the installation process for users. This capability also means that there is no need to ship a bootable disk with your software. Second, a user can run the Installer from an AppleShare server. The Installer can also use files located on an AppleShare server.

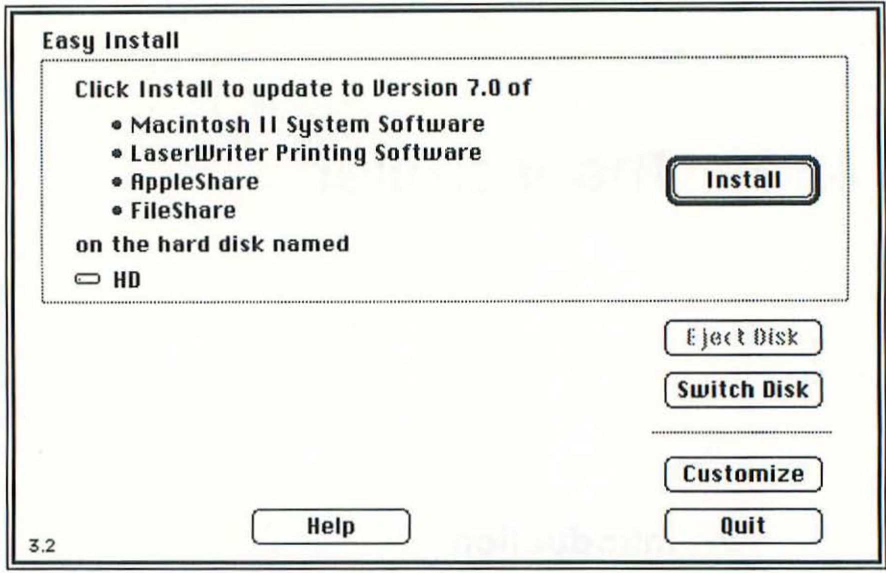


Figure 4-1. The one-button install dialog for System 7

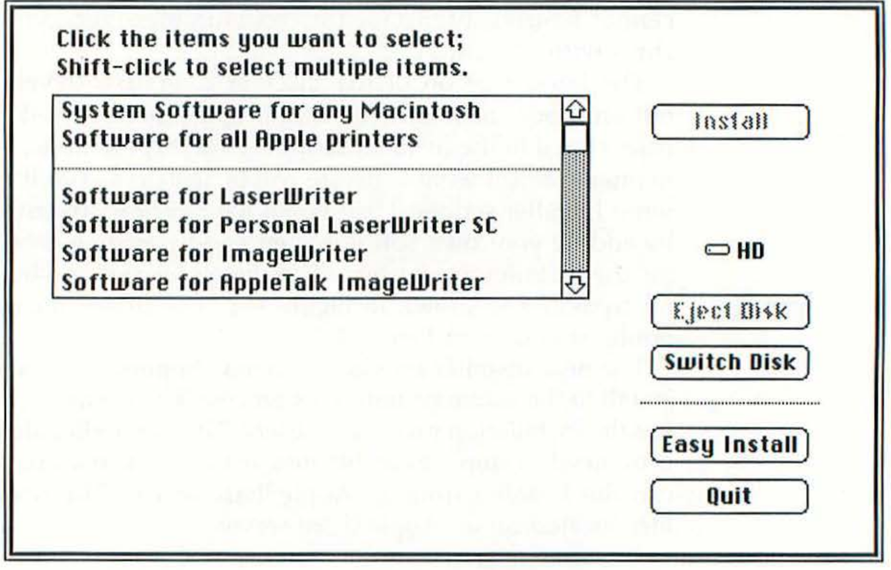


Figure 4-2. The optional install dialog for System 7

Note ▶ These two capabilities can reduce the administrative burden of installing software at sites with many Macintoshes.

The Installer provides a complex case study in how to take advantage of the Resource Manager. Several levels of resources can refer to other resources in Installer scripts. Installer resources are complex, but the tasks the Installer must accomplish are sometimes complex too.

▶ Using the Installer

The user has two ways to use the Installer: Easy Install or Custom Install. If the user selects Easy Install, then the Installer uses a set of rules in an Installer script to decide exactly what will be installed. You can write rules that can install different resources depending on which Macintosh the user is running on, for example. The standard Easy Install script for System 7 does this, instead of forcing users to select from a rather cryptic list of software to install.

If the user wants to control what software will be installed, then he or she presses the Custom Install button. This button presents a list of software to install. The choices for System 7 include a minimal system for each member of the Macintosh family, EtherTalk software, printer software, File Sharing, and so on. You cannot write rules to control the custom installation process.

▶ What the Installer is Really Doing

Whether the user selects Easy Install or Custom Install, the Installer will operate from a script. A *script* is a binary file containing resources unique to the Installer. Script files are built from a textual description of the script using the Rez MPW tool.

▶ Developing an Installer Script

The Installer follows a script when installing software. You write the script that tells the Installer what to do. The process of developing Installer scripts is illustrated in Figure 4-3. You start by creating a textual description of the script, perhaps using MPW. Rather than starting from scratch, you may find it easier to use another script as a base. Using MPW's DeRez tool, you can always create an .r file from another script.

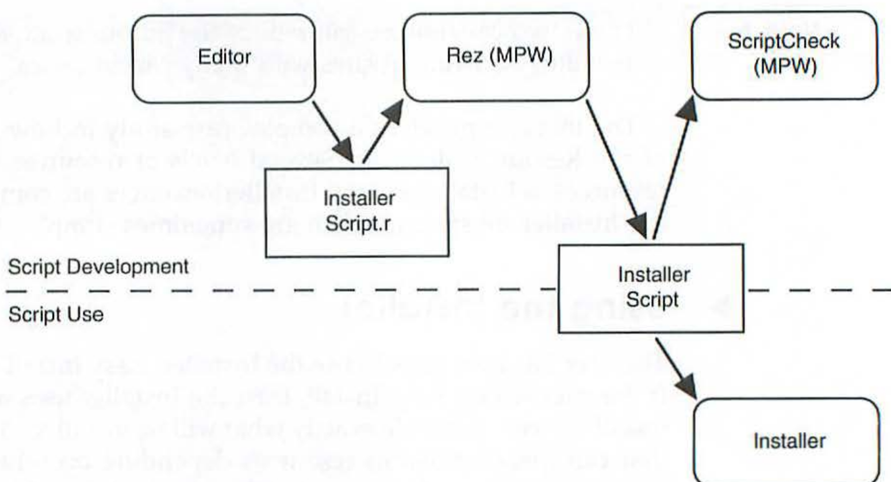


Figure 4-3. The development process for Installer scripts

After you've edited your script, run it through the Rez tool. This tool compiles the textual description of the resources and produces a file that matches the description. If Rez reports any errors, you should fix them before proceeding.

Following this, you'll run the script against the ScriptCheck MPW tool. This tool serves two purposes. First, it completes the script by handling some of the more tedious aspects of script writing. Examples of this include filling in the size fields of the Installer resources and recording file creation dates. Second, it checks for consistency and logic errors. This tool reports three levels of potential problems. *Notes* are the least severe and may not be problems; *Warnings* are the next category; and *Errors* are the most severe problems. You can also use ScriptCheck to remove any unused script resources in the script file. ScriptCheck also compares the disks that will be distributed to users against their Installer scripts.

▶ The Organization of an Installer Script

The Installer uses several kinds of resources. Before looking at the details of these resources, let's look at the way in which these resources, and therefore scripts, are organized. The relationship between the components of an Installer script is illustrated in Figure 4-4.

At the lowest level are *atoms*, of which there are five kinds: file atoms,

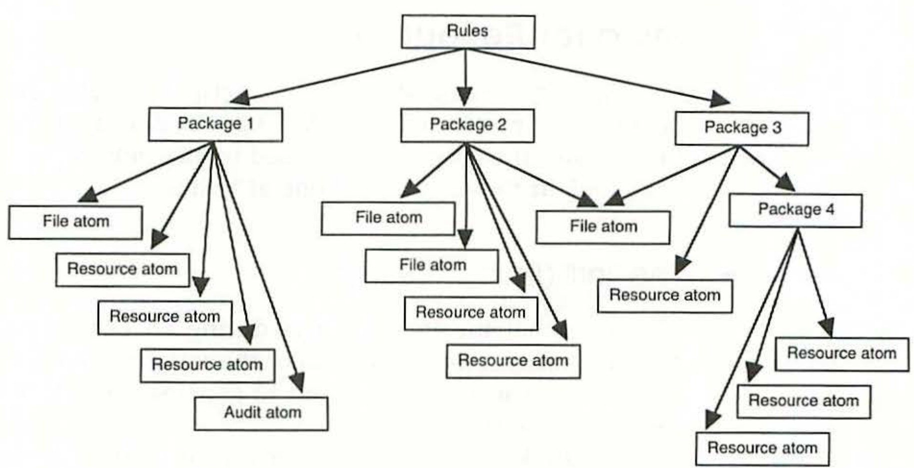


Figure 4-4. The components of an Installer script

resource atoms, action atoms, audit atoms, and boot block atoms. A file atom specifies a file to be copied or deleted. A resource atom specifies a resource from a particular file to be copied or deleted. An action atom is an external piece of code that can be executed as part of the installation process. An audit atom contains historical information about what has been installed. A boot block atom describes boot block parameters that should be changed or preserved, and when boot blocks should be written as part of the installation process.

Atoms are collected into *packages*. Actually, the documentation refers to packages and subpackages, but they are the same thing: a collection of atoms and other packages. In this chapter, I will refer to both packages and subpackages simply as *packages*. Use packages to collect related atoms to simplify the script. Users who choose Custom Install are actually installing packages. In Figure 4-4, notice that Package 4 is contained in Package 3. An atom or package can be contained in more than one package. In Figure 4-4, notice that one file atom is contained in both Package 2 and Package 3.

Packages are visible to users in the custom installation process. Atoms are not visible to users.

Packages and atoms can be managed by *rules*. Rules govern how the installation process happens when the user selects Easy Install. You can only have one set of rules per Installer script. If the script has no rules, then the user cannot use the Easy Install process.

▶ Installer Resources

Now that you understand how a script is organized, let's look at the details of scripts. Installer version 3.0 introduced a new series of resources that replace the 'insc' resource used by previous versions of the Installer. Let's look at these resources one at a time.

▶ The 'inrl' (Rule) Resource

The rules that the Installer uses during an Easy Install are specified in resources of type 'inrl'. The goal of the set of rules in a script is to generate a list of packages to install and to generate an informative message for users of the script.

Rules look like if-then statements in programming languages. For example, "*if* the current machine has an Ethernet card, *then* install the EtherTalk software." Rules are composed of clauses. The two kinds of clauses are conditional and generative. A conditional clause specifies a condition that must be met. A generative clause adds to either the list of packages to be installed or the user message. The various types of clauses are listed in Table 4.1. All clauses with a prefix of *check* are conditional clauses, and all clauses with a prefix of *add* are generative clauses. A third type of clause, reportError, allows your script to alert the user to any problems.

Table 4-1. The types of clauses allowed in Installer rules

Clause	*Parameters*	*Returns TRUE If:*
checkGestalt	Gestalt selector, list of acceptable return values	Gestalt returns a value from supplied list
checkMinMemory	Amount of memory (in Mb)	Amount of memory on current machine is $\geq$ minimal amount specified
checkFileDataForkExists	File specification	File exists and has a data fork
checkFileRsrcForkExists	File specification	File exists and has a resource fork
checkFileContainsRsrcByID	File specification, resource type, resource ID	File exists and has a resource of specified type and ID
checkFileContainsRsrcByName	File specification, resource type, resource name	File exists and has a resource of specified type and name

Table 4-1. The types of clauses allowed in Installer rules (continued)

Clause	*Parameters*	*Returns TRUE If:*
checkFileVersion	File specification, minimal version	File exists and has a 'vers' resource ≥ minimal version
checkFileCountryCode	File specification, country code	File exists and country code in 'vers' resource equals specified country code
checkTgtVolSize	Minimal size, maximal size	Selected volume has more than minimal space and less than maximal space; (0,0) matches any disk, (x>0,0) requires a minimum only
checkATVersion	Minimal version	AppleTalk version is ≥ specified version
checkUserFunction	User function type and ID	Extensibility hook; you can create your own functions to check other things and it returns TRUE or FALSE
addAssertion	List of 1 or more assertions	Always TRUE; used to set assertions to TRUE
checkAllAssertions	List of 1 or more assertions	All assertions are TRUE
checkAnyAssertions	List of 1 or more assertions	At least one of the assertions is TRUE
checkMoreThanOneAssertion	List of 1 or more assertions	More than one of the assertions is TRUE
addUserDescription	String	Always TRUE; used to add to the text that will be shown in the Easy Install dialog
addPackages	List of package IDs	Always TRUE; used to add a set of packages that will be installed with Easy Install
reportError	String	Always TRUE; used to display string to user in a caution dialog box
addAuditRec	File specification, selector, value	Always TRUE; used to add an Installer audit record with the specified selector and value to the file
checkAuditRec	File specification, selector, value	File has an audit record with the specified selector and value
checkAnyAuditRec	File specification, selector, list of acceptable values	File has an audit record with the specified selector and one of the specified values

The clause called **checkUserFunction** provides one way of customizing the Installer. Although the variety of clauses is quite broad, the standard clauses do not cover everything. You can use **checkUserFunction** to write a procedure that returns TRUE or FALSE to check on whatever you need.

A rule returns a logical value equal to applying the AND operator to the results of all the clauses in the rule together. A rule is said to "fire" if every clause returns TRUE.

▶ The 'infr' (Rule Framework) Resource

Resources of type 'infr' are used to describe a *rule framework.* A rule framework forces the rules specified in 'inrl' resources to be evaluated in a particular order. The Installer walks through the rule frameworks in order of resource ID. Each rule framework contains an ordered list of rule IDs and a command either to evaluate the list of rules until one of the rules fires, or to evaluate all the rules in the list.

▶ Assertions

Assertions provide a way of saving state information, that is, of saving the results of rules that have fired. You can use assertions to simplify a script. To use an assertion, you must first define a unique symbol. Then set an assertion by using the **addAssertion** clause in a rule. You can test if an assertion is set by using either the **checkAllAssertions** or **checkMoreThanOneAssertion** clause.

▶ The 'inpk' (Package) Resource

Resources of type 'inpk' are used to describe packages. A package is a set of one or more atoms and packages. Packages that have their ShowsOnCustom bit set in their resource are listed in the Custom Install dialog. They are displayed in the same order as their resource IDs.

If a package has its Removable bit set in its resource, then the Installer will permit the user to remove the package. If you hold down the Option key when the Custom Install dialog is displayed, then the Install button will change to Remove. If you do make a package removable, be careful of unwanted side effects. For example, your application might require a font to be installed that could also be used by some other application. Your script cannot assume that it is safe to remove that font because another installer script may also have installed the font.

The package resource contains a name that is displayed in the list of

packages in the Custom Install screen. The core of the 'inpk' resource is a "parts" list, which contains the resource type and ID for each component of this package. A package can contain any number of each of the five types of atoms as well as other packages. The package resource also contains a resource ID for an 'icmt' resource. The 'icmt' resource contains a comment that will be displayed when this package (and this package only) is selected by the user.

▶ The 'infa' (File Atom) Resource

Resources of type 'infa' are used to describe file atoms; that is, files that are to be copied or deleted during the installation process. One 'infa' resource is required per file.

The 'infa' resource contains flags specifying if the following conditions hold:

- The data fork of the source file will be copied (or deleted)
- The resource fork of the source file will be copied (or deleted)
- The copy operation should happen only if the target file already exists (that is, if this is an update-only file)
- The copy operation should not happen if the target file already exists
- The copy operation should not happen if the target file is newer than the source file (you might use this flag for Preferences files)
- The target file should be deleted before the source file is copied
- The target file is removable when the user clicks the Remove button

The 'infa' resource also contains the file specifications for the target and source files. Actually, it contains resource IDs for 'infs' resources for those files. Finally, the 'infa' resource contains a file size field and a description field, the latter of which is displayed by the Installer when it is copied.

▶ The 'inra' (Resource Atom) Resource

Resources of type 'inra' are used to describe resource atoms; that is, resources that are to be copied or deleted during the installation process. One 'inra' resource is required per resource.

The 'inra' resource contains flags equivalent to those of an 'infa' resource, as well as flags for the following:

- Use the name or resource ID to find the resource
- Whether the name should match

The 'inra' resource contains the resource type and source and target resource IDs. The 'inra' also contains the file specifications for the target and source files related to this resource. Actually, it contains resource IDs for 'infs' resources for those files. Lastly, the 'inra' resource contains a resource size field and a description field, the latter of which is displayed by the Installer when it is copied.

If you are working with fonts, then be careful to specify them using only 'FOND' resources, and not 'FONT' or 'NFNT' resources. The user might otherwise end up with inconsistent font information. Specify the font you want and the sizes you want in the name field only; don't use the resource IDs. For example, use

```
Palatino 9 10 12 14
```

to copy the Palatino font along with the bitmap resources for the point sizes 9, 10, 12, and 14. The Installer handles all of the associated font resources automatically. All styles are copied automatically.

▶ The 'inaa' (Action Atom) Resource

Resources of type 'inaa' are used to identify a piece of code in the script file that can be executed as part of the installation process. One 'inaa' resource is required per piece of code. This code is executed when the Installer has completed all other activity, just before returning to the user.

The 'inaa' resource contains the resource type and resource ID of the code to be executed. It also contains a description field that is displayed by the Installer when the code is being executed. A pair of flags, actOnInstall and actOnRemove, cause the code to be executed when these flags are set.

▶ The 'inat' (Audit Atom) Resource

Resources of type 'inat' are used to describe audit atoms. These atoms can cause resources of type 'audt' to be added to the target file. If the 'audt' of that ID already exists, then its contents are overwritten with the new value.

This resource provides a way to customize the installation based on the options used for previous installations. You would use the **checkAudit-Record** or the **checkAnyAuditRecord** clause in your script to do this.

The 'inat' resource contains the resource ID of a file specification ('infs')

for the target and an array of selectors and values. The selectors are usually resource types, and the values are a 4-byte integer.

▶ The 'inbb' (Boot Block Atom) Resource

Resources of type 'inbb' are used to describe changes that should be made to the target volume's boot blocks. One 'inbb' resource is required for each boot block parameter specified. You can change each field in the boot blocks by specifying its new value.

You can also have a new set of boot blocks written to the target volume. In this case, you would enter the resource ID of the 'infs' for the file containing the boot blocks.

▶ The 'icmt' (Installer Comment) Resource

Resources of type 'icmt' are used to contain Installer comments. The Installer displays these comments when the user has first chosen Custom Install and has then selected one package to install. Installer comments have four fields: a version release date, a version number, an icon (actually the 'icmt' resource only contains a resource ID for an 'ICON'), and a string. The resource ID of the 'icmt' is contained in the 'inpk' resource.

▶ The 'infs' (File Specification) Resource

Resources of type 'infs' are used to specify files in a symbolic manner in Installer scripts. These resources are used by rules and atoms to refer to specific files.

The 'infs' resource contains the file type and creator, a creation date, and a pathname. You can also use some flags to specify whether the file type and creator of the specified file should match those in this resource and whether the file should be searched for if it isn't found at the specified path.

The pathname for a source file contains the complete pathname of the file, starting from the volume on which the file exists. The pathname for a target file contains a modified partial pathname, because the target volume is not known until long after the script has been written. You can specify the System Folder or one of the special folders in it by using a pathname starting with *special-* followed by the resource type of the folder. For example, to specify that the target is a file called MyPreferences, which should be installed in the Preferences folder, you would give the pathname as

```
special-pref:MyPreferences
```

▶ The 'indo' (Disk Order) Resource

Resources of type 'indo' are used to specify the names and order in which disks should be used in the installation process. By default, disks are requested in the order in which atoms require them. With larger scripts, you can make the installation process easier and less confusing by controlling the order in which disks are requested.

▶ Customizing the Installer

You have already seen two methods for customizing the Installer. First, you can specify an action atom (a resource of type 'inaa') in a script. This will cause a code resource in the script to be executed after all the files and resources have been copied and deleted by the Installer. Second, you can write a user function that can be called in a clause in a rule. Such a function returns TRUE or FALSE. You can also use a third method to customize the Installer: Create a splash screen for a script.

▶ Creating a Splash Screen

A *splash screen* is a resource 'PICT' that is displayed when the Installer starts executing a script. Create a 'PICT' resource by copying a picture drawn in a paint or draw program to the clipboard. Start ResEdit, create a new file, and paste from the Clipboard. Delete all resources except for the 'PICT' resource. Change the name of this 'PICT' to *Splash Screen*. You can include this file in your Installer script by using an include (Rez) statement.

▶ When to Use the Installer

Apple designed the Installer because a tool with sophisticated capabilities was needed to install the system software. The Installer has always allowed users to update their system software without disturbing the fonts, desk accessories, and other resources previously installed.

The Installer is not the only tool available for installing software. StuffIt is an example of another software installation tool. These alternate tools provide some form of file compression, which is their primary advantage. On the other hand, none of them can install resources into a file. Neither can they install desk accessories, INITs, fonts, and so on. When is the Installer the better choice and when should you use another tool?

Use the Installer when your software package

- Contains files that must be installed in more than one folder

- Contains resources that must be installed in one or more files
- Includes one or more desk accessories, drivers, INITs, sounds, and/or fonts
- Should be installable across a network
- Should be installable live; that is, installable as the package is running on the machine

Use another installation tool when your software package

- Is installed in a single folder
- Can be compressed to fit onto a single floppy disk

Don't use any installation tool when

- Your software package includes only a small number of files that all belong in a single folder
- All the files fit onto a single floppy disk

▶ Conclusion

In this chapter, you've looked at the new Installer. It is a much more powerful and flexible tool than previous versions. You can customize the Installer if you require something not yet supported.

Get Info ▶

For more information on the Installer, refer to the *Installer Technical Reference Guide*, the *Macintosh Installer ScriptCheck User's Manual*, and the *Installer Script-Writing Hints & Tips*.

5 ▶ Compatibility

▶ Introduction

One reason why the Macintosh has gained (and maintained) popularity is the high degree of compatibility between members of the Macintosh family and between versions of the Macintosh operating system. This contrasts with both MS-DOS and Windows, where many applications have required major modifications to run on newer versions of these environments.

In this chapter, you'll first look at the Gestalt Manager. The Gestalt Manager replaces the **Environs** and **SysEnvirons** system calls from earlier versions of the Macintosh operating system. The **Gestalt** call provides all the features of these earlier calls and a lot more.

Next, you'll look at compatibility with A/UX, Apple's version of the UNIX operating system. Almost all of the Macintosh operating system is available under A/UX, and so almost all Macintosh applications should be able to run in this environment. You'll learn how to make this happen.

Last, you'll look at what it takes for your application to take best advantage of System 7. System 7 brings important new capabilities to the operating system. Some of the capabilities will quickly become as ubiquitous as cut, copy, and paste. You'll look at which capabilities are in this category.

The issue of providing support for other countries and other languages in not covered in this chapter. See Chapter 9 for information on international services.

The Gestalt Manager

The Macintosh operating system becomes more complex with each release, especially with System 7. The Gestalt Manager was added (starting with version 6.0.4) because programmers were having a difficult time writing code to figure out whether a particular feature was available in the operating system.

Trying to decide whether an operating system feature is present by looking at the ROM version, the CPU type, or even the version of the operating system is risky. Newer versions of the operating system can patch new versions of system calls into older machines and older ROMs. As new versions of the operating system are introduced after your application has shipped, some changes to the operating system can invalidate the assumptions you made when writing your application. **Gestalt**, the primary call in the Gestalt Manager, provides a way to figure out whether an operating-system feature exists.

Calling Gestalt

Before you call **Gestalt**, you need to make sure that the system call is implemented on the current machine. If you're using MPW version 3.2 or later, you don't have to worry about whether the **Gestalt** call is available. This is because MPW adds some "glue code" to the libraries that handle calls to **Gestalt** when running on versions of the operating system.

Gestalt is easy to call—you need to pass it a *selector*, which is a code for the kind of information you'd like to know and the address of a long integer for the results. **Gestalt** also returns an error code.

The two kinds of selectors are environmental and informational. Use the environmental selectors to decide whether a feature is present. Table 5-1 lists all the environmental selectors available for use with the 7.0 version of **Gestalt**.

Table 5-1. Environmental selectors for Gestalt

Selector Name	*Selector*	*Information Returned*	*Type*
gestaltAddressingModeAttr	'addr'	Addressing mode attributes	attributes
gestaltAliasMgrAttr	'alis'	Alias Manager attributes	attributes
gestaltAppleEventsAttr	'evnt'	AppleEvents attributes	attributes
gestaltAppleTalkVersion	'atlk'	AppleTalk version	version
gestaltAUXVersion	'a/ux'	A/UX version, if present	version

Table 5-1. Environmental selectors for Gestalt (continued)

Selector Name	*Selector*	*Information Returned*	*Type*
gestaltConnMgrAttr	'conn'	Connection Manager attributes	attributes
gestaltCRMAttr	'crm '	Communication Resource Manager attributes	attributes
gestaltCTBVersion	'ctbv'	Communications Toolbox version	version
gestaltDBAccessMgrAttr	'dbac'	Database Access Manager attributes	attributes
gestaltDITLExtAttr	'ditl'	Dialog Manager extensions	attributes
gestaltEasyAccessAttr	'easy'	Easy Access attributes	attributes
gestaltEditionMgrAttr	'edtn'	Edition Manager attributes	attributes
gestaltExtToolboxTable	'xttt'	External Toolbox trap table base	address
gestaltFindFolderAttr	'fold'	FindFolder attributes	attributes
gestaltFontMgrAttr	'font'	Font Manager attributes	attributes
gestaltFPUType	'fpu '	Floating-point unit type	type
gestaltFSAttr	'fs '	File system attributes	attributes
gestaltFXfrMgrAttr	'fxfr'	File Transfer Manager attributes	attributes
gestaltHardwareAttr	'hdwr'	Hardware attributes	attributes
gestaltHelpMgrAttr	'help'	Help Manager attributes	attributes
gestaltKeyboardType	'kbd '	Keyboard type	type
gestaltLogicalPageSize	'pgsz'	Logical page size	size
gestaltLogicalRAMSize	'lram'	Logical RAM size	size
gestaltLowMemorySize	'lmem'	Size of low-memory area	size
gestaltMiscAttr	'misc'	Miscellaneous information	attributes
gestaltMMUType	'mmu '	Memory management unit type	type
gestaltNotificationMgrAttr	'nmgr'	Notification Manager attributes	attributes
gestaltNuBusConnectors	'sltc'	NuBus connector bitmap	attributes
gestaltOSAttr	'os '	Operating-system attributes	attributes
gestaltOSTable	'ostt'	Operating-system trap table base	address
gestaltParityAttr	'prty'	Parity attributes	attributes
gestaltPhysicalRAMSize	'ram '	Physical RAM size	size
gestaltPopupAttr	'pop!'	Popup CDEF attributes	attributes
gestaltPowerMgrAttr	'powr'	Power Manager attributes	attributes
gestaltPPCToolboxAttr	'ppc '	PPC Toolbox attributes	attributes
gestaltProcessorType	'proc'	Processor (CPU) type	type
gestaltQuickDrawVersion	'qd '	QuickDraw version	version

Table 5-1. Environmental selectors for Gestalt (continued)

Selector Name	*Selector*	*Information Returned*	*Type*
gestaltResourceMgrAttr	'rsrc'	Resource Manager attributes	attributes
gestaltScriptCount	'scr#'	Number of active script systems	count
gestaltScriptMgrVersion	'scri'	Script Manager version	version
gestaltSerialAttr	'ser '	Serial hardware attributes	attributes
gestaltSoundAttr	'snd '	Sound attributes	attributes
gestaltStandardFileAttr	'stdf '	Standard File Package attributes	attributes
gestaltStdNBPAttr	'nlup'	Standard Name Binding Protocol attributes	attributes
gestaltTermMgrAttr	'term'	Terminal Manager attributes	attributes
gestaltTextEditVersion	'te '	TextEdit version	version
gestaltTimeMgrVersion	'tmgr'	Time Manager version	version
gestaltToolboxTabl	'tbtt'	Toolbox trap table base	address
gestaltVersion	'vers'	Gestalt Manager version	version
gestaltVMAttr	'vm '	Virtual memory attributes	attributes

Table 5-2 lists all the informational selectors. These selectors should *not* be used to answer questions like, "Does the current machine have a memory management unit?" That's why the environmental selectors are provided. Rarely, you may need to use the informational selectors for this purpose, but this shouldn't happen often. Use the informational selectors to retrieve information to display to users.

Table 5-2. Informational selectors for Gestalt

Selector Name	*Selector*	*Information Returned*	*Type*
gestaltMachineIcon	'micn'	Machine icon	icon
gestaltMachineType	'mach'	Machine type	type
gestaltROMSize	'rom '	ROM size	size
gestaltROMVersion	'romv'	ROM version	version
gestaltSystemVersion	'sysv'	System version	version

The results returned by **Gestalt** are of a particular type. The result type is described by a suffix on the selector name and is listed in the last column of Tables 5-1 and 5-2. The results should be interpreted according to Table 5-3. Note that the results might not use all four bytes. If this is the case, then the lower-order bytes are used to store the results.

Table 5-3. Gestalt return types

Result Type	*Interpretation*
attributes	The results should be interpreted as a string of bits. Each bit must be interpreted individually, although not all bits are used for a selector. Bit 0 is the least significant bit.
count	The results are a count of how many of that item exist.
icon	The result is a 32 x 32 icon.
size	The results are the size of the value requested in bytes.
type	The results are an index into a list of types.
version	The results are a version number. The first two bytes are the major version number, and the last two bytes are the minor version number.

▶ Modifying the Gestalt Manager

Two additional system calls are provided so that you can modify the responses of the Gestalt Manager. You can change the response to an existing selector or add a procedure to respond to a new selector. Most applications will not need to do this, but certain classes of applications will find these calls useful.

You can change the response to an existing selector by using the **ReplaceGestalt** call. You need to pass the selector type and the address of a function that will handle the selector. If this call succeeds, it returns the address of the old selector function. You'll need this address if your new selector function needs to call the old function or if you need to restore the old selector function.

You can add support for a new selector by using the **NewGestalt** call. You need to pass it the selector type and the address of a selector function. If the selector is already supported by the Gestalt Manager, you'll get an error. You'd use **NewGestalt** when you're writing an engine that provides application-independent services. You could register the existence of these services by using this call. You could also describe its attributes using this call. This would provide a simple way to test for its existence, the current version number, and the attributes of the service.

By the Way ▶

If you've registered your application's signature with Apple, you can use your application's creator type as the selector type. Apple has reserved for itself all sequences consisting of four lowercase letters and all sequences of nonalphabetic characters.

Selector Functions

A selector function takes a selector type and returns its results in a long integer (whether or not all four bytes are required). Selector functions can call other selector functions including, if you're using the **ReplaceGestalt** call, a previous selector function. You can't use globals in the A5 world unless you specifically set them up and restore the previous values when you're done.

Link the selector function into a 'GDEF' resource. The attributes for this resource should specify that it should be locked and loaded into the system heap. Your code will need to intall the 'GDEF' resource into the system heap. An example of this is provided in the Compatibility Guidelines chapter of *Inside Macintosh,* Volume VI. Note that once your selector function is installed in the system heap, it will be there after your application quits until the machine is restarted. Your selector function should handle this situation.

Running under A/UX

A/UX is Apple's version of the UNIX operating system. A/UX version 2, in addition to providing all the basic features of UNIX, provides several features unique to Apple, such as a Macintosh-style window for the command shells, MPW Commando-style interfaces to all the UNIX commands, and, most important, access to most of the Macintosh operating system. The latter feature was a component of A/UX version 1.0, but A/UX version 2 provides a high degree of compatibility with the Macintosh operating system.

Unless you have a good reason to avoid A/UX support, you should try to ensure that your applications can run under A/UX. The market for A/UX will be growing rapidly now that version 2 has been released. The government and higher education are the two market segments expected to accelerate A/UX's growth.

You have two ways to use the Macintosh operating system and Toolbox under A/UX. First of all, most Macintosh applications can potentially run under A/UX. They behave as they do under the Macintosh operating system. With version 2 of A/UX, several Macintosh applications can be run simultaneously because a version of MultiFinder comes with A/UX. As it turns out, you don't have to do much to most Macintosh applications to make them compatible with A/UX. Let's look at the two aspects of A/UX compatibility: the availability of components of the Macintosh operating system under A/UX, and programming techniques to ensure compatibility.

Second, UNIX applications written specifically for A/UX can make calls to the Macintosh operating system. This allows UNIX applications to provide a Macintosh human interface that looks and feels like an application running under the Macintosh operating system.

▶ Availability of the Macintosh Operating System under A/UX

The version of the Macintosh operating system and Toolbox available for use while running in the A/UX environment is known as the *A/UX Toolbox*. The A/UX Toolbox implements user-interface Toolbox calls primarily by using the routines in the Macintosh ROMs. Calls to the Macintosh operating system, as opposed to the Toolbox, are primarily emulated by translating these calls into their UNIX equivalents. Fortunately, this process is transparent to application programmers.

The one aspect of using the Macintosh operating system that is not transparent to application programmers is that not all of the calls are available under A/UX. The availability of the various managers is described in Table 5-4.

Table 5-4. Availability of managers under A/UX version 2

Availability	*Manager*
Full implementation	AppleTalk Manager
	Binary-Decimal Conversion Package
	Color Manager
	Color Picker Package
	Color QuickDraw
	Control Manager
	Desk Manager
	Device Manager
	Dialog Manager
	Disk Driver
	Disk Initialization Package
	File Manager
	Font Manager
	Gestalt Manager
	International Utilities Package
	List Manager
	Memory Manager
	Menu Manager

Table 5-4. Availability of managers under A/UX version 2 (continued)

Availability	*Manager*
	Notification Manager Package Manager Palette Manager Printing Manager QuickDraw Resource Manager Scrap Manager Script Manager Serial Driver Slot Manager Sound Manager Startup Manager Standard File Package TextEdit Toolbox Utilities Window Manager
Full implementation (all calls available), but the behavior of the call may be different than under the Macintosh operating system	Event manager (Toolbox portion) Floating-point arithmetic Shutdown Manager System Error Handler Time Manager Transcendental Functions Package
Partial implementation (most calls available)	Event Manager (Operating System portion) Segment Loader Utilities, Operating System Vertical Retrace Manager
None	Alias Manager Apple Desktop Bus Data Access Manager Deferred Task Manager Edition Manager Help Manager Power Manager PPC Toolbox SCSI Manager

It is important to remember when using the Macintosh managers under A/UX that the version of the manager may be different from the latest version of that manager available under the Macintosh operating system. This leads to the next topic: programming for A/UX compatibility.

▶ Programming Techniques for A/UX Compatibility

What do you have to do to ensure that your application can run under A/UX? In addition to the standard programming techniques for compatibility under the Macintosh operating system, five rules are especially important:

1. Use the Gestalt Manager to check whether the versions of the managers you want to use are available. Implementation of managers under A/UX is usually a version or two behind the Macintosh operating system.

By the Way ▶ A/UX version 2 provides managers equivalent to those available under version 6.0.5 of the Macintosh operating system.

2. Make sure your code is 32-bit clean (see Chapter 16). A/UX is a 32-bit environment, so code that is not 32-bit clean will crash under A/UX.
3. Make sure that your application is MultiFinder-friendly. MultiFinder is running all the time under A/UX version 2, just as it is under System 7. Therefore, you should be calling **WaitNextEvent** and not **GetNextEvent**. You should also provide a 'SIZE' resource to describe the application.
4. Avoid using low-memory globals. Some low-memory globals are not supported at all under A/UX. The use of low memory has been discouraged by Apple for some time, but on some occasions you have no choice but to read from or write to low-memory globals when writing Macintosh applications.
5. Don't talk to hardware directly. Use managers and device drivers to talk to hardware. A/UX uses memory protection, whereas the Macintosh operating system does not. A/UX will not allow you to talk to memory-mapped input/output devices—your application will crash if you try this.

▶ System 7-Aware Applications

System 7 allows three classes of applications: 7.0-compatible, 7.0-dependent, and 7.0-friendly. An application is *7.0-compatible* if it can run under System 7. The majority of current Macintosh applications fall into this

category because they were developed using the Apple's programming guidelines.

An application is *7.0-dependent* if it requires System 7 or later. An application that requires the Data Access Manager would be 7.0-dependent because this manager didn't exist before System 7.

Finally, an application is *7.0-friendly* if it takes advantage of certain new features available with System 7. A 7.0-friendly application meets the following conditions:

- Supports MultiFinder—Your event loop should use the **WaitNext-Event** call instead of **GetNextEvent**. This has not changed from System 6, but MultiFinder is always running under System 7, so now you *must* be compatible with MultiFinder.
- Supports the standard Apple events—Interapplication communication facilities are available under System 7, and Apple is strongly encouraging all applications to support the high-level messages known as Apple events. The Finder sends Apple events to Apple event-aware applications to open and print files, for example. See Chapter 6 for more information about Apple events.
- Runs under virtual memory—Most applications are not affected by virtual memory, but a handful of applications may need to make calls to lock data structures in memory to ensure reasonable performance. See Chapter 16 for more information about virtual memory.
- Has no restriction on font size—System 7 introduces a new font technology called TrueType, which is implemented in the Font Manager and allows for arbitrary (integral) point sizes of fonts. The older font technology required a bitmapped version of each size for optimal display on the screen, but TrueType can generate a version for any size in real time. See Chapter 8 for more information about TrueType.

Supporting some of these features will require little or no work for most applications. Other features will require more extensive work.

Applications that take the most advantage of System 7 are said to use the "Magnificent 7." Such applications

- are 32-bit clean
- are MultiFinder-aware
- have no restrictions on font size

- support the Edition Manager
- support the required and core AppleEvents
- provide Balloon help
- are AppleShare-aware

▶ Conclusion

In this chapter, you've looked at three aspects of compatibility. First of all, you looked at the Gestalt Manager, which provides a simple, standard way of checking version numbers (of the operating system, of managers, and so on) and checking whether certain operating-system features are supported. By using the Gestalt Manager, you can be assured of compatibility with future versions of the Macintosh operating system.

Next, you looked at what it takes to make an application compatible with A/UX. For most applications, little additional effort is required beyond following standard programming guidelines.

Last, you looked at what it takes to make an application 7.0-friendly. Apple is encouraging all applications to support a subset of new features introduced (or improved) with System 7.

Get Info ▶

For more information about 32-bit cleanliness, read Chapter 16 in this book, the Memory Manager chapter, *Inside Macintosh*, Volume VI, and Technical Note #212.

Read the Compatibility Guidelines chapter of *Inside Macintosh*, Volume VI for more information about the Gestalt Manager.

For more information on Macintosh operating-system support under A/UX, read *A/UX Toolbox: Macintosh ROM Interface*, which is one of the manuals that comes with A/UX.

6 ▶ The PPC Toolbox, High-Level Events, and Apple Events

▶ Introduction

In this chapter, you'll look at some new features of the Macintosh operating system: Apple events, high-level events, and the PPC Toolbox. The PPC Toolbox, which is the foundation on which these other services are built, provides a set of low-level interapplication communications (IAC) services. These services do not differ substantially in function from the IAC services provided by other operating systems, such as UNIX. You'll first look at what the PPC Toolbox is and how you can use it.

Next, you'll move on to high-level events, which use the PPC Toolbox for support and hide a lot of the plumbing. Because of this, you'll find high-level events easier to use than the PPC Toolbox.

Last, you'll look at Apple events, which are a specific class of high-level events. These events are being standardized, and many applications will be using them. The Apple event protocols will be supported as widely as are cut, copy, and paste by applications we have today.

▶ The PPC Toolbox

The PPC Toolbox provides Program-to-Program Communications (PPC) services—low-level services that are called interapplication communications (IAC) services on other operating systems. You can use the PPC Toolbox to transfer data to and from other applications in real time. The pair of communicating applications can reside on a single Macintosh or on two Macintosh computers on a network. From a programmer's point of view, it's as easy to use the PPC Toolbox over a network as it is to use locally.

▶ PPC Terminology

The PPC services enable your software to communicate with other applications through a *port*. A port has a name and a location: The *port name* must be unique on its machine, and the port location identifies the machine on the network. Each application that *uses the PPC services* must start up a *session* with the other port involved. To do this, both applications must have opened a port. Your application can send and receive *blocks* of data to and from another port. The blocks of data have no standard format.

The italicized words in the previous paragraph are technical terms you should avoid in documentation for users. To avoid using computer jargon in software and manuals, Apple recommends using the terminology listed in Table 6-1.

Table 6-1. Preferred user terminology for PPC Toolbox concepts

Programmers Say:	*In English:*
executable code	program
uses PPC Toolbox	supports program connections
PPC session	program connection
end a PPC session	disconnect the programs

▶ Naming PCC Ports

Port names have two formats. The first format consists of a name string followed by a type string, as in

```
MyDatabase,database
```

You must use a comma to separate the two strings. The second format consists of a name string followed by a 4-byte Macintosh creator type and the 4-byte port type (the latter is similar to file types, but applies to ports).

Port locations consist of an object string, followed by a type string, followed by the AppleTalk zone. You must also use some special punctuation. An example of a port location is

```
TonyM:PPCToolbox@BearRiver
```

where *TonyM* is the object string (and is usually the owner name as entered in the Sharing Setup control panel), *PPCToolbox* is the type string, and *Bear-*

River is the AppleTalk zone. This format is defined by Name Binding Protocol (NBP), which is one of the protocols in the AppleTalk protocol suite.

Note ▶

Future versions of the PPC Toolbox will have other formats for location name. Another method of naming locations would be to use a name server on the network. This would simplify the maintenance of names and addresses, because you could look up an address by knowing a user name. Users could then move from one location to another without having to notify everyone of their new network address. Apple has not discussed this alternative, but it is an obvious direction to go in. Other networks and network operating systems are heading in this direction.

▶ When to Use the PPC Toolbox, When to Use Apple Events

You need to be aware of the tradeoffs in using the PPC Toolbox, on the one hand, and high-level events and Apple events, on the other hand. You should probably be using the PPC Toolbox when you're writing one of the following:

- Something other than an application, such as a device driver
- A server that requires sharing resources across multiple sessions
- Code that is not driven by events
- Code that must run asynchronously

▶ Compatibility and the PPC Toolbox

Call **Gestalt** with a selector of **gestaltPPCToolboxAttr** to verify that the PPC Toolbox is available and can support real-time delivery of messages. (See Chapter 5 for more information on **Gestalt**).

▶ Managing PPC Services

Each user has control over the usage of PPC services on his or her machine. By using the Sharing Setup control panel (cdev), the user can enable or disable the usage of network PPC services (and also Apple events, which you'll learn about later in this chapter). The user cannot control the local usage of PPC services.

When a remote user wants to use the PPC services of this user's Macintosh, the local user must enable these services. The local machine will reject the session request from the remote user if the PPC services are not enabled.

The local Macintosh must also authenticate the remote user. *Authentication* means that the remote user must provide a user name and password that correspond to an entry in the local Macintosh's Users and Groups control panel. Each entry in this folder controls the access of a remote user, although the folder has a default entry for "guests." This default entry can allow nonregistered users to have some access to the services on the local machine. If the remote user cannot provide an acceptable user name and password, and cannot be signed on using the guest account, then the session request will be rejected. Authentication is not available for local use of the PPC Toolbox.

Important ▶ The maintenance of user names and passwords must be done locally on each machine. System 7 has no provisions for a network-based name service, which would enable a network manager to handle this burdensome task. Although this is in keeping with the Macintosh way of doing things, large organizations may have some problems maintaining access control.

▶ Calling the PPC Toolbox

Let's now look at the routines in the PPC Toolbox. Call **PPCInit** to initialize the PPC Toolbox. After making this call, you can call any of the other PPC Toolbox routines synchronously or asynchronously. You should make most calls to the PPC Toolbox routines synchronously. However, you should usually call **PPCInform**, **PPCRead**, and **PPCWrite** asynchronously. This improves the performance of your software. When you do call a PPC Toolbox routine, you'll need to pass the address of a *completion routine*. The PPC Toolbox calls the completion routine when your call completes, whether it was successful or not. You should use completion routines to finish off whatever processing is required following the completion of the call.

Most PPC Toolbox calls require a *parameter packet*. The contents of the parameter packet are different for each call. This chapter covers only some of the parameters for each call. For the details, refer to the PPC Toolbox chapter of *Inside Macintosh*, Volume VI.

The first time an application wants to communicate with another application using the PPC Toolbox, the user will probably need to tell the

application which other application he or she wants to use. Call **PPCBrowser** to present the user with a standard dialog box for this purpose. This dialog box shows the user a list of zones (if the user's Macintosh is located on an internet), a list of Macintosh computers, and a list of applications. When making this call, you supply a prompt for the dialog box, a title for the list of PPC ports (the default is Applications), and an optional default PPC port. You can also supply a port filter function, which this dialog will call for each PPC port before adding it to the list. The filter function returns TRUE if the PPC port should be displayed in the browser, and FALSE otherwise.

Important ▶ No standards exist for naming services now and there may never be any. Therefore, beware of this when writing a filter function. For example, port names and locations are case-sensitive. Names and locations can also be in other scripts. (For more information on scripts, see Chapter 9.)

For occasions when you don't need or want to present the PPC browser, you should call **IPCListPorts**. This routine returns a list of all ports visible to the network. You supply the maximum number of ports that you want to look at and a pointer to a buffer that will hold the list of names. You can also pass a template for the port names that you'd like. If you pass NIL for this parameter, you'll get a list of all the ports on the local machine. By using the NBP metacharacters, you can get a list of all the ports in your network, or all ports with the same name on your network. Other variations are also available.

When should you use **PPCBrowser**, and when should you use **IPCListPorts**? Obviously, if you don't know the name and location of the port you want talk with, you probably want to call **PPCBrowser**. If your application will be communicating with that port regularly—for example, if it will be using a particular server in future sessions—you could save the name and location of the port. When the application is started again, you could use **IPCListPorts** to check if that port is open. You would also use **IPCListPorts** when your software doesn't have an interface. In this case, your software will presumably know what port to look for.

Call **PPCOpen** to open a PPC port. When making this call, you specify the port's name and location and whether the port should be visible on the network. **PPCOpen** will return a port reference number if the call was successful. You'll need this number to make certain other calls to the PPC Toolbox.

In the current implementation of the PPC Toolbox, all sessions happen in real time, so applications must be running simultaneously in order to use the PPC Toolbox.

Note ▶

In future versions of the PPC Toolbox, a store-and-forward mode will also be available. This will allow one application to leave messages for another application that may not be running at the same time.

To initiate a PPC session with another port, you'll call either **PPCStart** or **StartSecureSession**. In either case, you'll supply the port reference number of the port you're communicating from, the name and location of the PPC port you want to hold the session, whether this session is to be connected in real time, and four bytes of user data. This data is sent to the other port and is obtained there by a call to **PPCInform**, which is discussed below. If this call to either **PPCStart** or **StartSecureSession** is successful, you'll get a session reference number that uniquely identifies this session. If the attempt to start a session was rejected, you'll get a reject code (which is different from the error code returned by this call).

StartSecureSession provides authentication services to verify that the user has the appropriate privileges on the other end of the session. This call is equivalent to asking for the user's name and password, and then calling **PPCStart**. You can pass the user's name and a prompt for the standard dialog box requesting the user's name and password. **StartSecureSession** encrypts the user name and password and sends them to the remote machine for authentication. The remote machine, which must have already called **PPCInform**, can accept or reject the request. A dialog box is displayed if the request was rejected.

Call the **PPCInform** routine if your application can receive session requests. You should make this call after you've opened a PPC port. You'll need to supply the port reference number when calling **PPCInform**. You can specify whether to automatically accept session requests. Make sure this call is executed asynchronously, or your application will be idle until a session request arrives. You can have more than one open call to **PPCInform**.

When a session request arrives, following your call to **PPCInform**, you'll get the user name, the name and location from which the request came, the user data provided by the requestor, and whether the request came from the local machine or from somewhere else on the network.

When you receive a request for a session, following your call to

PPCInform, you should call either **PPCAccept** or **PPCReject**. Note that a call to **PPCReject** is not the same as a rejection where the user couldn't be authenticated: You might reject a call because you don't have sufficient resources at this time or because you're about to shut down and don't want any new sessions to begin.

Once a session is initiated, use the **PPCRead** and **PPCWrite** calls to receive and send blocks of data. These calls behave much like file I/O calls. You need to specify parameters such as a buffer pointer and buffer length in addition to such PPC-specific parameters as session reference number. Calls to **PPCRead** and **PPCWrite** should usually be executed asynchronously. Apple recommends that once you've started a session, you should always have a pending **PPCRead** to provide quick notification in case of an error.

A block of data can contain information in any form, since the PPC Toolbox simply treats it as a string of bytes. Blocks can be as large as 2^{32} bytes. You can specify a creator and type for the first block to be written, which is helpful if your software will be working with more than one type of data. The creator and type are passed to the receiver of the message.

To end a PPC session, call **PPCEnd** whether you used either **PPCStart**, **StartSecureSession**, or **PPCAccept** to start the session. You must specify the session reference number of the session you wish to end.

When you're all done with a port, call **PPCClose** to shut it down. You'll need to pass the port reference number you got when the port was first opened.

Two calls are useful for dealing with user authentication. Call **Get-DefaultUser** to get the user reference number and name of the default user. Call **DeleteUserIdentity** to remove the name and password for the specified user reference number.

▶ An Example of Using the PPC Toolbox

Let's now look at an example of how you might use the PPC Toolbox. This example is illustrated in Figure 6-1. In this illustration, time flows down the diagram—that is, a higher call is made before the call(s) below it on that machine. The remote machine hosts a server that provides a service that other applications on the network need to use. In this example, the server maintains a real-time database of stock values. When it gets a stock name, it returns the latest value for that stock. The user at the local machine often needs to look up the values of various stocks.

The server software, as part of its initialization process, calls **PPCOpen** to open a PPC port, and then **PPCInform** to let others know that the

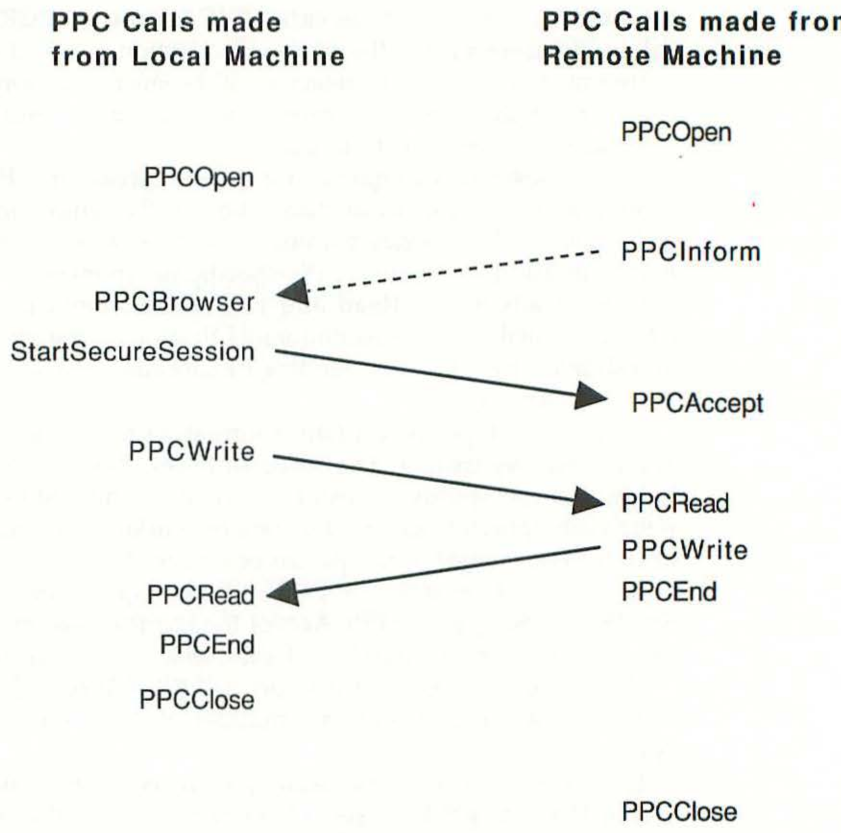


Figure 6-1. Example of using the PPC Toolbox

server is available. At the local machine, the user starts the stock-quote software and specifies the server through the dialog box displayed by the call to **PPCBrowser**. When the user needs a stock quote, he or she logs onto the server by calling **StartSecureSession**. The server gets the request, verifies the user name and password, and then accepts the request with the call to **PPCAccept**. Next, the server, which is written to use a particular protocol, issues a call to **PPCRead** to get a stock name.

The stock-quote software on the local machine then calls **PPCWrite** to send the name of the stock and issues a call to **PPCRead** to get the results back from the server. Following the completion of the call to **PPCRead**, the server software gets the name. Using a call to **PPCWrite**, the server looks up the value of the stock and returns the value. The **PPCRead** call from the stock-quote software is then completed, and the value is dis-

played to the user. You close the session with a call to **PPCEnd**, and close the port with a call to **PPCClose**.

Meanwhile, the server has issued a call to **PPCEnd** to close this session. When the server shuts down, it makes a call to **PPCClose** to close the PPC port.

In reality, the stock-quote software probably wouldn't close the session after getting a single quote. It would start a session with the server and continue the session, with many writes and reads, before closing the session.

▶ Using High-Level Events

Let's now look at what high-level events are and how to use them. Before we do that, let's review the role of events in the Macintosh operating system.

▶ Types of Events

Macintosh applications are said to be *event-driven* because the basic structure of the application is a loop of code that gets an event and then processes it. System 7 allows three categories of events in the Macintosh operating system:

- Low-level events
- Operating-system events
- High-level events

▶ Low-Level Events

Some low-level events tell the application the following information:

- About actions performed by the user (mouse down, mouse up, key down, key up, key autorepeat, disk inserted)
- About activity from other sources (device driver)
- A message from the application itself
- That there are no other events to report—that is, the null event

The Operating System Event Manager (separate from the Toolbox Event Manager) maintains a single queue of these low-level events. These events are sent to the active application.

Other low-level events report changes in the position of windows. The activate, deactivate, and update events are generated by the Window Manager and are sent to the Toolbox Event Manager.

Low-level events (and all other events) are described by an Event-Record, as illustrated in Figure 6-2. The event code, the first field in this data structure, describes the kind of event.

what (event code)
message
when (clock ticks since startup)
where (mouse location)
modifiers (flags)

Figure 6-2. Structure of an EventRecord

▶ Operating-System Events

Operating-system events were introduced with MultiFinder. The operating system sends these events to tell the application about changes in its status. The kinds of operating-system events are suspend, resume, mouse-moved, and application-died.

A *suspend* event tells your application that it should prepare to move from the frontmost position. It should hide floating windows, unhighlight selections, convert the local scrap to the global scrap, and so on. Your application is not suspended until it makes another call to **WaitNext-Event**, **GetNextEvent**, or **EventAvail**. Once it is suspended, your application will not get any CPU time until it becomes the frontmost application, unless it can perform some useful activity in the background.

A *resume* event tells your application to become the frontmost application. It should restore any highlighting, restore floating windows, convert the global scrap to the local scrap if needed, and so on. As soon as the application's state has been restored, it then proceeds with the regular event loop processing.

Mouse-moved events are sent to applications that have specified a region in their call to **WaitNextEvent**. You use this call primarily to adjust the cursor shape.

Application-died events were used primarily for debuggers. These events are now sent as Apple events.

▶ High-Level Events

High-level events are new with System 7 and provide a high-level inter-application communications service. These services are built on top of the PPC Toolbox. The operating system automatically handles PPC session management and buffering. High-level events are easier to use than the PPC Toolbox, but high-level events are primarily intended to allow a pair of applications to exchange messages. The facilities of the PPC Toolbox are more general and more powerful, but require more work. High-level events can be used easily over a network, just like the PPC Toolbox.

The operating system maintains a separate queue for high-level events for each application that has described itself as capable of handling them. The size of these queues is limited by available memory.

For the most part, high-level events are handled as yet another case in the regular event loop processing. A couple of new calls have been added to the Event Manager to provide access to the optional data associated with high-level events. These calls are described later in this chapter.

▶ Describing High-Level Events

High-level events are described differently than other types of events, although you obtain them by using one of the regular three calls to the Event Manager: **WaitNextEvent**, **GetNextEvent**, or **EventAvail**. High-level events are identified by a particular mask set in the event record.

The meaning of some fields in the EventRecord is different for high-level events. The message field and the where field have different meanings here than for low-level or operating-system events.

The *message field* contains the message class, which identifies the message *protocol*. To allow applications to communicate, the applications must be able to understand a common set of messages, known as a protocol. The Event Manager does not understand protocols, but it provides a way for a message to be identified as a message from a particular protocol.

Apple events have a message field containing 'aevt' and are the most important protocol because they standardize a wide range of high-level events for all Macintosh applications. You'll look at Apple events later in this chapter following the survey of high-level events.

Another protocol is the one used by the Edition Manager (with a message field of 'sect'). High-level events are sent by applications that publish a new version of a document and every other time an event happens that will affect another application using that edition. This protocol can be treated as a set of high-level events, or it can be more easily handled as Apple events.

If you have registered your application creator type with the Developer Technical Support group at Apple, you can use your creator type as the message class for a private set of high-level events. If you define a private set of events, only those applications that know about them can use these events.

The *where field* contains the message ID. This defines a particular message in the message class, such as cut or copy in the Apple events class.

The message and where fields provide a complete set of information for some messages in a message class. At times, this is not enough information; in these cases, a high-level event may require additional data. Also, it is often important to know which application sent the event. This additional information is not contained in the EventRecord. Several new routines in the Event Manager provide access to this information for applications.

▶ Calling High-Level Event Manager Routines

To get any additional information about a high-level event after calling **WaitNextEvent**, **GetNextEvent**, or **EventAvail**, call **AcceptHighLevel-Event**. You need to pass the address of a message buffer and the address of a TargetID data structure, which contains the relevant session reference number and the port name and location of the sender. The **AcceptHigh-Level** call fills the buffer with as much data as is available (or will fit in the buffer) and the number of bytes of data in the buffer. If there is more data than can fit in the buffer, the buffer will contain as much data as will fit and you'll get a BufferIsSmall error code. In this case, you can call **Accept-HighLevelEvent** again to get the rest of the data.

If, as part of your protocol, you need to retrieve a specific high-level event from the high-level event queue, call **GetSpecificHighLevelEvent**. You'll need to pass the address of a filter procedure and a set of search parameters. The filter function checks each record in the high-level event queue until either it finds a record that matches your criteria or there are no more records.

To send a high-level event to another application, call **PostHighLevel**. The parameters for this call include an event record, who the message is going to, the (optional) address to a message buffer, and delivery options. You can specify where the message is going in several ways. If the message is being sent to a process on another machine, you can specify the process by session ID or port name and address. If the receiver is on the local machine, you can also specify the process by process serial number or signature.

Delivery options include giving this message priority and requesting a return receipt. Message priority is simply a flag; the receiving application

will have to look for a high-level message with this flag set to process priority messages before nonpriority messages.

Return receipts are posted by the Event Manager on the receiver's machine, not by the receiver itself. The receipt is simply another high-level message (of class 'jaym' and message ID 'rtrn'). The modifiers field of the event record tells you whether the message was accepted. A message can also be partially accepted—that is, some, but not all, of the data in the message buffer may have been read by the receiver. If you need to know who the return receipt is from (because you have more than one message out at a time), call **AcceptHighLevelEvent** and look in the sender parameter. This contains the identity of the application that "sent" the return receipt.

Two utility routines convert port names to and from process serial numbers: **GetProcessSerialNumberFromPortName** and **GetPortNameFromProcessSerialNumber**. These two routines are useful only for translating local process serial numbers—they will not return valid information for remote process serial numbers. For more information on process serial numbers, refer to Chapter 17.

▶ The 'SIZE' Resource

The 'SIZE' resource was introduced with MultiFinder. MultiFinder uses an application's 'SIZE' resource to determine the size of the memory partition to allocate for the application, as well as whether the application is MultiFinder-aware. Under System 7, the 'SIZE' resource has been extended to describe additional characteristics of the application, such as whether it can accept local high-level events and remote high-level events, and whether it can handle stationery documents directly.

System 7 has integrated all the functions of MultiFinder directly into the operating system. The 'SIZE' resource describes certain capabilities of the application so that the operating system can provide the optimal level of services for it. Every application should have a 'SIZE' resource, or else the system will use a default set of parameters that produce less than optimum performance.

The 'SIZE' resource contains 16 bits of flags and two 32-bit integers. The two integers hold the sizes of the minimum and preferred memory partitions. The minimum partition size is the smallest memory partition size in which your application will run. The preferred size is the largest partition size that your application would like. Your application will be launched if at least enough memory for the minimum partition is available. The user will be asked to confirm that he or she wants to start the application if less memory than the preferred partition size is available.

The flags describe how the operating system should deal with the application. Some of these flags have been added with System 7. These flags are described in detail in the Event Manager chapter of *Inside Macintosh*, Volume VI. The following flags are new since System 7:

- isStationeryAware—If set, means that your application recognizes stationery documents. If this bit is not set, the Finder duplicates the document and prompts the user for a name.
- RemoteHLEvents—If set, means that your application will be visible to other applications over the network. If this flag is not set, your application won't get any high-level events over the network. This flag does not affect local high-level events.
- isHighLevelEventAware—If set, means that your application can deal with high-level events. If it is not set, then your application will not receive any high-level events.
- useTextEditServices—If set, means that your application wants to use the inline text input services of TextEdit.

▶ Apple Events

Apple events are a specific high-level event protocol called the Apple Event Interprocess Messaging Protocol (AEIMP). This protocol is a standard for all applications. Your application should implement all the Apple events that are applicable to your class of application so as to be as compatible with as many other applications as possible.

Apple events are divided into several categories. The most important category is the required Apple events. Any application that claims to support Apple events must support this minimal set. Only four events are in the set of required Apple events: open application, open documents, print documents, and quit.

These required Apple events are important because the Finder will send these Apple events to an Apple event-aware application. The Finder will use the older mechanism (which requires using the **CountAppFiles** and **GetAppFiles** routines) for applications that cannot handle Apple events. The Finder can tell that your application is Apple event-aware if the isHighLevelEventAware flag is set in the 'SIZE' resource.

Other categories of Apple events are grouped into successively more specialized categories. Some sets of Apple events are for basic manipulation of text and graphics, and other sets are more specialized for various categories of applications. These sets of Apple events will evolve, with more events and more categories of Apple events being standardized over time. By supporting standard sets of Apple events, applications will

become more open. Users will be then able to use their applications in more flexible and effective ways.

Apple events are handled like high-level events, as previously described. You need to add the case of high-level events to your event loop. The Apple event Manager handles much of the work needed to process Apple events. This means that Apple events are simpler to deal with than high-level events in general. In turn, high-level events are simpler to deal with than the PPC Toolbox, on which Apple events are based. Apple events can be used easily over a network, just like the PPC Toolbox and high-level events.

Note ▶ Future versions of Apple events may use some transport mechanism other than high-level events. This shouldn't affect any applications that are written to work with Apple events, because the Apple event Manager presents a very high-level interface to these functions. Any changes to the Apple event transport mechanism should be transparent to applications that use the Apple event Manager.

▶ Compatibility and Apple Events

Call **Gestalt** with a selector of gestaltAppleEventsAttr to verify that Apple events are supported on the current machine.

▶ Important Data Structures of the Apple Event Manager

First, here is some terminology used to describe Apple events. An Apple event is sent by a *source* to a *target*. A transaction involving a set of Apple events is started by a *client application* that uses the services of a *server application*. A transaction typically requires more than one Apple event. Note that the source of an Apple event is always the sender of that event, which means that for a given event the client application can act as either the source or the target at a given point in the transaction. Correspondingly, the server application will act as the target or the source.

An Apple event consists of a set of *Apple event attributes* and *Apple event parameters*. Every application uses the services of the Apple Event Manager to create, access, and dispose of these attributes and parameters. You will never directly access either the attributes or parameters of an Apple event.

The attributes of an Apple event specify the task to be performed using the data in the parameters. Attributes include the event class, event ID, and target application.

Apple event parameters can be divided into several categories. *Direct parameters* contain data to be used in performing the task specified by the attributes of an Apple event. Most direct parameters are *required parameters*, that is, parameters that must be provided as part of the Apple event. Other parameters may be *optional parameters*, that is, parameters for which defaults are specified. Last, some parameters may be additional parameters, that is, parameters that will be used in addition to the data specified in the direct parameters.

The fundamental data structure used by the Apple Event Manager is the AEDesc or *descriptor record*. A descriptor record consists of a descriptor type, a 4-byte code that describes the data type contained in the descriptor record, and a handle to the data. Table 6-2 lists the various descriptor types.

Table 6-2. Apple event descriptor types

Category	*Descriptor Type*	*Value*	*Data Type*
Apple event	typeAppleEvent	'aevt'	Apple event record
	typeAEList	'list'	List of descriptor records
	typeAERecord	'reco'	List of keyword-specified descriptor records
	typeType	'type'	Four-byte code for event class or event ID
	typeKeyword	'keyw'	Apple event keyword
	typeProperty	'prop'	Apple event property
	typeTargetID	'targ'	Apple event target ID record
Text	typeChar	'TEXT'	Unterminated string
Number	typeSMInt	'shor'	16-bit Integer
	typeInteger	'long'	32-bit Integer
	typeMagnitude	'magn'	Unsigned 32-bit integer
	typeSMFloat	'sing'	SANE single precision number
	typeFloat	'doub'	SANE double precision number
	typeExtended	'exte'	SANE extended precision number
	typeComp	'comp'	SANE comp number
	typeEnumerated	'enum'	Enumerated data
Boolean	typeBoolean	'bool'	Boolean value
	typeTrue	'true'	TRUE boolean value
	typeFalse	'fals'	FALSE boolean value
System	typeAlias	'alis'	Alias record
	typeApplSignature	'sign'	Application signature
	typeAppParameters	'appa'	Process Manager launch parameters
	typeFSS	'fss '	File system specification (FSSpec) record
	typeProcessSerialNumber	'psn '	Process serial number
	typeSectionH	'sect'	Section record handle

Table 6-2. Apple event descriptor types (continued)

Category	*Descriptor Type*	*Value*	*Data Type*
	typeSessionID	'ssid'	Session ID
	typeTemporaryID	'tid '	Temporary ID
Other	typeNull	'null'	NULL or nonexistent data
	typeWildCard	'****'	Matches all types

An *address descriptor record* (AEAddressDesc) has the same structure as an AEDesc record. They differ in that an AEAddressDesc contains the address of the source (sender) or target (receiver) of an Apple event.

An *keyword-specified descriptor record* (AEKeyDesc) also has the same structure as an AEDesc record except that an AEKeyDesc contains an Apple event keyword in place of a descriptor type. Apple event keywords are listed in Table 6-3. Notice that while descriptor types merely specify the data type, keywords specify a meaning for the data as well as the data type.

A *descriptor list* (AEDescList) has the same structure as an AEDesc record. They differ in that an AEDescList always contains data of type

Table 6-3. Apple event keywords

Category	*Keyword*	*Value*	*Data Description*
Attribute keyword	keyAddressAttr	'addr'	Address of the target application
	keyEventClassAttr	'evcl'	Event class of the Apple event
	keyEventIDAttr	'evid'	Event ID of the Apple event
	keyEventSourceAttr	'evrc'	Source of the Apple event
	keyInteractLevelAttr	'inte'	User interaction level for server application
	keyMissedKeywordAttr	'miss'	Remaining required parameter in an Apple event
	keyOptionalKeywordAttr	'optk'	List of optional parameters for an Apple event
	keyReturnIDAttr	'rtid'	Return ID for any reply Apple events
	keyTimeoutAttr	'timo'	Number of clock ticks that the client process will wait for a reply from server
	keyTransactionIDAttr	'tran'	Transaction ID (identifies a set of associated Apple events)
Parameter keyword	keyDirectObject	'----'	Direct parameter
	keyErrorNumber	'errn'	Error number parameter
	keyErrorString	'errs'	Error string parameter
	keyProcessSerialNumber	'psn '	Process serial number parameter

typeAEList, that is, a list of descriptor records. An AERecord is the same as an AEDescList, but is used in a different context. An *Apple event record* (Apple event) is defined to be an AERecord except that it is always of type typeAppleEvent. An Apple event record contains a list of attributes and parameters, whereas an AERecord contains only parameters. You can optionally pass an Apple event record in place of an AERecord when using the routines of the Apple Event Manager. Notice that you cannot do the reverse, that is, you cannot pass an AERecord in place of an Apple event (Apple event record).

Apple event records, AEDesc, and the other data structures bear a strong resemblence to LISP data structures. They contain lists, which can contain other lists, and so on. The names of the some of the Apple event Manager routines also reflect a LISP-like approach (such as AEGetNthPtr).

▶ Using Apple Event Dispatch Tables

Create an Apple event dispatch table for your application by calling **AEInstallEventHandler** for each Apple event that your application can handle. You'll supply parameters including the event class, the event ID and a pointer to an event handler routine for that type of Apple event. This dispatch table is used in a similar manner to the trap dispatch table, which is used to handle Macintosh system calls. The Apple event dispatch table is clearly operating at a more abstract level.

Call **AEGetEventHandler** to get a pointer to the handler for the specified Apple event belonging to the specified class. Call **AERemoveEventHandler** to remove an Apple event handler for the specified Apple event belonging to the specified class.

Call the **AEProcessAppleEvent** routine, perhaps in your main event loop, to process the specified Apple event. This system call first looks in the application's Apple event dispatch table for a handler for this event. If a handler has been installed there, then control is passed to it. If a suitable handler does not exist there, then this system call looks in the system Apple event dispatch table for a handler. If a suitable handler has been installed there, then control is passed to it. If no handler is found for this event, then it returns an errAEEventNotHandled to the server application and, if requested, to the client application.

Rather than passing the current Apple event as a parameter to your routines, your routines can use the **AEGetTheCurrentEvent** system call. This will reduce the number of parameters needed for your routines and will make an incremental performance improvement in them.

Your application can suspend and resume processing an Apple event

by using the **AESuspendTheCurrentEvent** and **AEResumeTheCurrentEvent** system calls. The Apple Event Manager does not dispose of Apple events automatically, so you can suspend and resume the processing of an Apple event without having to worry about storing the event.

You can also take advantage of special handler dispatch tables. Handlers of this type are called before the Apple event handler dispatches an Apple event. There is an application Apple event dispatch table for each application and a system Apple event dispatch table. If you install a special handler in the system table, then it will be called for Apple events for all applications. Under System 7, the only type of special handler which can be installed is of type keyPreDispatch.

Call **AEInstallSpecialHandler** to install a handler for the specified keyword in the specified special handler dispatch table. Call **AEGetSpecialHandler** to get a pointer to the handler for the specified keyword from either the application or system Apple event dispatch table. Call **AERemoveSpecialHandler** to remove the handler for the specified keyword from either the application or system Apple event dispatch table.

You can write your own Apple event handlers. Your handler should use Apple event Manager routines to extract the attributes and parameters from an Apple event. It should then perform the task specified by the attributes.

▶ Extracting Data from an Apple Event

The Apple Event Manager provides a full set of routines for extracting the attributes and parameters from an Apple event. These routines can similarly be used to extract data from descriptor records, AERecords, and other data structures similar in structure. Apple event Manager routines that have a suffix of **-Ptr** extract data from a descriptor record into a buffer which you specify. Apple event Manager routines which have a suffix of **-Desc** extract descriptor records. Table 6-4 lists the routines for extracting data.

The Apple Event Manager allows you to coerce data to other data types. Call AECoercePtr or AECoerceDesc to coerce data to the specified type. You can install your own coercion routines in a coercion handler table. There are three routines for manipulating this table: **AEInstallCoercionHandler**, **AEGetCoercionHandler**, and **AERemoveCoercionHandler**. These routines behave similarly to the corresponding Apple event handler routines.

A very important routine, used when sending or receiving Apple events, is **AEDisposeDesc**. The Apple event Manager does not automatically dispose of any descriptor records. That means that *any* time that you

Table 6-4. Routines for extracting data from Apple event data structures

Routine Name	*Function*
AEGetAttributePtr	Extracts data for the specified attribute
AEGetAttributeDesc	Extracts descriptor record for the specified attribute
AEGetParamPtr	Extracts data for the specified parameter
AEGetParamDesc	Extracts descriptor record for the specified parameter
AEGetNthPtr	Extracts data for the *n*th item of a parameter's list
AEGetNthDesc	Extracts descriptor record for the *n*th item of a parameter's list
AECountItems	Counts the number of items in a descriptor list
AEGetArray	Converts an Apple event array into a C or Pascal array
AEGetKeyPtr	Extracts data from the specified keyword-specified descriptor record
AEGetKeyDesc	Extracts descriptor record from the specified keyword-specified descriptor record
AESizeOfAttribute	Returns size and descriptor type of the specified attribute
AESizeOfParameter	Returns size and descriptor type of the specified parameter
AESizeOfNthItem	Returns size and descriptor type of the *n*th descriptor record in a descriptor list
AESizeOfKeyDesc	Returns size and descriptor type of the specified keyword-specified descriptor record

have created a descriptor record of any kind, you are responsible for disposing of it when you are done with it. This one routine disposes of a descriptor record or any of its derivative types.

▶ Sending an Apple Event

The Apple Event Manager provides a full set of routines for creating an Apple event or other Apple Event Manager data structures.

Call **AECreateAppleEvent** to create an Apple event and **AECreateDesc** to create a descriptor record. To duplicate an existing descriptor record, call **AEDuplicateDesc**. To create a descriptor list or AERecord, call **AECreateList**. Once you have called one of these routines to create a data structure, use the routines listed in Table 6-5 to add data to them or delete data from them.

Table 6-5. Routines for adding and deleting data from Apple event data structures

Routine Name	*Function*
AEPutPtr	Converts data from a buffer into a descriptor record and adds it to a descriptor list
AEPutDesc	Adds a descriptor record to a descriptor list
AEPutArray	Puts an Apple event array into a descriptor list
AEPutKeyPtr	Puts data into a keyword-specified descriptor record and adds it to an AERecord
AEPutKeyDesc	Converts descriptor record into a keyword-specified descriptor record and adds it to an AERecord
AEPutParamPtr	Converts data into a parameter and adds it to an Apple event
AEPutParamDesc	Converts descriptor record into a parameter and adds it to an Apple event
AEPutAttributePtr	Converts data into an attribute and adds it to an Apple event
AEPutAttributeDesc	Converts descriptor record into an attribute and adds it to an Apple event
AEDeleteItem	Deletes a descriptor record from a descriptor list
AEDeleteKeyDesc	Deletes a keyword-specified descriptor record from an AERecord
AEDeleteParam	Deletes an Apple event parameter

Call **AESend** to send the specified Apple event. Among the many parameters for this routine, you specify whether you want a reply Apple event or not, and whether you are willing to give up control of the processor while waiting for the reply. The latter option is especially useful when the source and target of the Apple event are on the same machine. Another parameter to specify is whether the server application should never, can, or will always interact with the user in response to this Apple event. You can also specify a timeout interval if you are waiting for a reply or expecting a return receipt.

Call **AESetInteractionAllowed** to control the user interaction preferences when responding to Apple events. There are three choices: Allow the server application to interact with the user

- only when the client and server application are the same
- only if the client application is on the same machine as the server application
- anytime

Call **AEGetInteractionAllowed** to get the current user interaction preferences. Call **AEInteractWithUser** before interacting with the user, such as displaying a dialog box or an alert. Remember that your application may be in the background at this time.

Call **AEResetTimer** to reset the timeout value for an Apple event to its initial value. Do this when you cannot reply before the client's application will time out.

▶ Where Are We Headed?

By using Apple events, high-level events, and the PPC Toolbox, you can create large applications that are more robust and more configurable. Applications will become more robust because a single application will no longer have to hold all the code. Such applications are hard to develop—witness the long time it has taken to ship some of the more complex word processing applications on the Macintosh. These applications are also difficult to maintain, simply because complex applications have a lot of code.

You can implement specialized features as separate, cooperative applications, which can be distributed separately from the primary application. The main application communicates with these specialized applications using Apple events, high-level events, or the PPC Toolbox. What would have been a single, complex application can now be separated into a primary and one or more secondary applications, each of which is simpler. In addition, when you use a well-defined protocol for communicating between them, the connections between these applications become simpler than what would be possible with a single application. The net effect is to reduce the overall complexity of the code, which decreases the time needed to design, implement, and test the code.

One architecture that will now be possible with Apple events will be to separate an application into an engine, which performs some set of functions and has little or no user interface, and a front-end, which provides the user with access to the functions of the engine. Applications will become more configurable because Apple events provides a simple method of communicating with applications.

Note ▶ Using AppleScript, a future scripting language, users will be able to configure their applications to better suit their needs.

These new features—Apple events, high-level events, and the PPC Toolbox—point toward where the Macintosh operating system is evolving. Rather than becoming like a traditional minicomputer operating sys-

tem, such as UNIX or VAX/VMS, the Macintosh operating system is growing toward a more object-oriented model. In this model, the boundaries between applications are permeable by messages. Users and other applications can send messages to an application to perform a task. Users can use their software by connecting them as components in various ways. Prior to System 7, applications could not communicate in a standard, high-level manner. Using the new features of System 7, tomorrow's software will be exciting to develop and to use.

▶ Conclusion

In this chapter, you've looked at the PPC Toolbox, which provides a low-level set of services for interapplication communications. High-level events provide a more abstract set of services, which are easier to use than the PPC Toolbox. Apple events are a standardized class of high-level events that will be supported by all applications, just as cut, copy, and paste are today.

Apple events are the first step in the evolution of applications and the Macintosh operating system. Rather than providing just low-level IAC services, Apple has provided a more sophisticated and powerful IAC mechanism built on top of the low-level services. In the long run, if you use these services, your applications will become more robust and more intelligent.

Get Info ▶

For more information on the PPC Toolbox, refer to the PPC Toolbox chapter of *Inside Macintosh,* Volume VI. For more information on the Name Binding Protocol, refer to Chapter 14 of this book, the AppleTalk chapter of *Inside Macintosh,* Volume VI, and *Inside AppleTalk.*

For more information on the high-level events, refer to the Event Manager chapter of *Inside Macintosh,* Volume VI. You may also want to refer to the Event Manager chapters of *Inside Macintosh,* Volumes I and V.

For more information on Apple events and the Apple Event Manager, refer to the Apple Event Manager chapter of *Inside Macintosh,* Volume VI, and to the APDA document, *The Developer's Guide to Apple Events.* For information on specific Apple event protocols, refer to the Technical Notes, which are widely available through APDA and on many electronic information systems.

7 ▶ The Edition Manager

▶ Introduction

In this chapter, you'll look at an important new feature of the Macintosh operating system: the Edition Manager. The Edition Manager provides a simple way to transfer data from one application to other applications and to automatically keep the data updated.

This new feature of the operating system provides an important and powerful set of features that are strongly analogous to the Clipboard. In fact, for the user, the Edition Manager will be like using copy and paste, except for three new commands named "Create Publisher...," "Subscribe to...," and "Publisher/Subscriber Options...".

To see how useful the services of the Edition Manager are, consider the following example. Suppose a work group is creating a proposal that contains, in addition to the text, numerous diagrams and pictures. The proposal is being written using a word processor, the diagrams are being created with a drawing program, and the pictures are being scanned. Naturally, various people are reviewing the proposal, and it must be updated until everyone is satisfied. The document could be created using cut, copy, and paste to move the diagrams and pictures. Unfortunately, each time a diagram or picture is changed, someone must go through the following steps:

1. Open the document containing the revised diagram
2. Select the diagram
3. Copy the diagram to the Clipboard

4. Switch to the word processor
5. Open the proposal
6. Find the old diagram
7. Select the old diagram
8. Paste the new diagram from the Clipboard, overwriting the old diagram
9. Save the document

This is an easy process to go through once, but if the proposal contains dozens or hundreds of illustrations, this becomes quite tedious.

Contrast that process with what you can do using the services of the Edition Manager:

1. Create the diagram in the drawing program
2. Select the diagram
3. Choose the "Create Publisher..." command
4. Switch to the word processor
5. Open the proposal
6. Place the cursor where the illustration should go
7. Choose the "Subscribe to..." command
8. Save the document

Thereafter, each time the illustrator changes the diagram, the word processor is notified of that change. It reads in the revised diagram and replaces the older version with the latest. Clearly, this will reduce much of the tedium involved in maintaining large or complex documents.

In this chapter, you'll first look at the user's view of publishing and subscribing, including the recommendations of Apple's Human Interface Group. Then you'll look at the new system calls you'll be using to implement these services. Last, you'll look at how to use these new calls.

▶ The User's View of Publishing and Subscribing

Inside Macintosh defines a *publisher* as "a section within a document that makes its data available to other documents or applications." It also defines a *subscriber* as "a section within a document that obtains its data from other documents or applications."

Let's look at the Clipboard before going through the features of the Edition Manager, because publishing and subscribing editions is similar

to using the Clipboard with copy and paste. The user selects some data and then chooses "Copy" from the Edit menu. After switching to another application, the user then chooses the "Paste" command from the Edit menu. By using this sequence of actions, the user has moved data from one application to another. The Clipboard and the commands in the Edit menu have always been part of the Macintosh operating system.

The basic difference between the Clipboard and the Edition Manager is that the Edition Manager can notify the subscribers of the edition whenever the publisher updates the edition. The Clipboard, on the other hand, must be reused each time to accomplish this task. This means that users can now be spared much of the drudgery of maintaining complex documents. Figure 7-1 shows the new standard commands that should be used in applications supporting the Edition Manager.

Create Publisher...
Subscribe to...
Subscriber options...
Show Borders

Figure 7-1. The standard commands for using the Edition Manager

The equivalent of the "Copy" command is "Create Publisher...." The user selects this command after making a selection. Next, a dialog box allows the user to name the edition file where the selected data will be stored, as well as the location where the file will be created. This dialog box is shown in Figure 7-2. Notice the area where the publishing application can display a small version of the data. This dialog box provides an example of how you can customize the Standard File Package.

When the same, or another, user wants to use the edition in a document, the user chooses the "Subscribe to..." command. This command presents the user with another dialog box, shown in Figure 7-3, which allows the user to navigate the file system to locate the edition file to which he or she wants to subscribe. Notice the area where the subscribing application can display a small version of the data. This dialog box provides another example of how you can customize the Standard File Package.

By default, every time the user saves a document that contains one or more publishers, the edition file(s) are updated. Also by default, every time the edition file is updated, the subscribing document is automatically

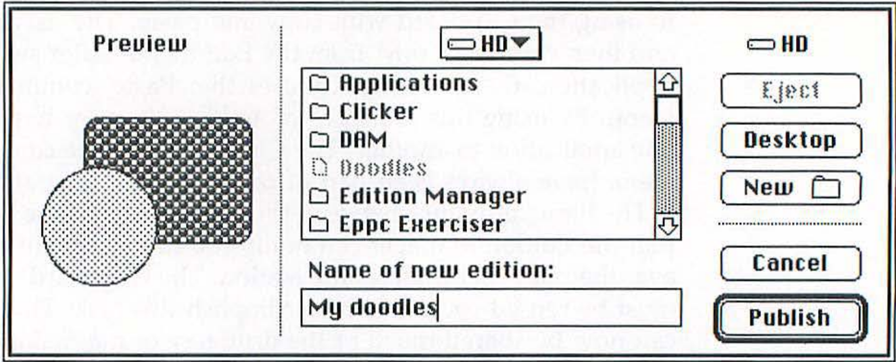


Figure 7-2. The standard dialog box for the "Create Publisher..." command

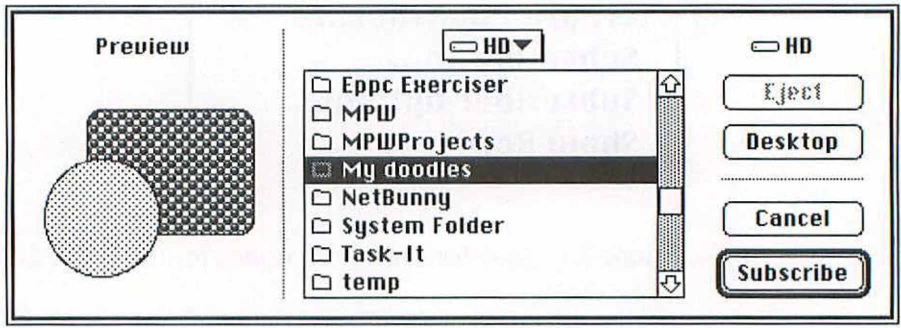


Figure 7-3. The standard dialog box for the "Subscribe to..." command

updated. The user controls these behaviors through the "Publisher Options..." and "Subscriber Options..." dialogs. These two dialogs are shown in Figures 7-4 and 7-5 respectively.

Each edition should have only a single publisher, or source of data. You can modify the content of a publisher. One publisher cannot overlap with any other publisher. The edition, or section, should be updated whenever the user saves the document that contains the publisher. When the section is updated, the Edition Manager notifies any subscribers that it has changed.

An edition can have none, one, or more than one subscriber. Subscribers should not be modified, but their presentation can be. This means that you can allow users to resize or reshape the subscriber, or if the subscriber is text, to change the style of the entire subscriber to bold. If the user does

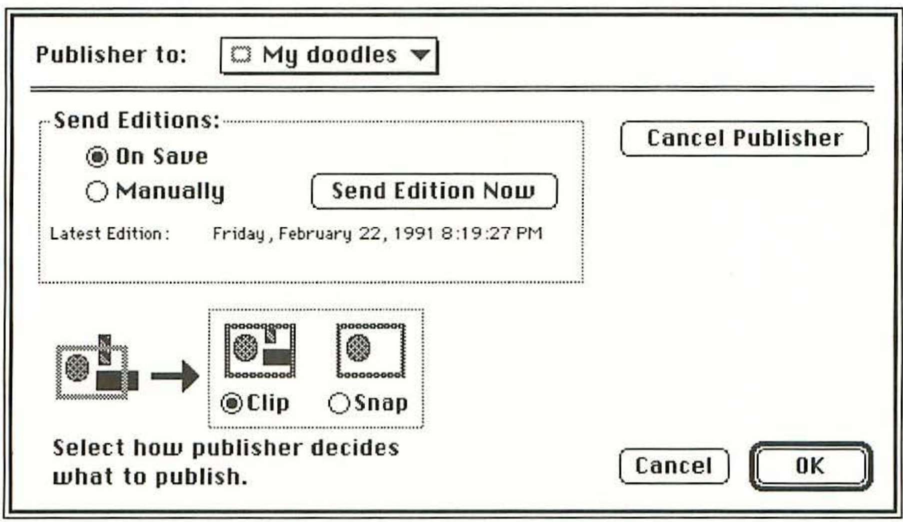


Figure 7-4. The standard dialog box for the "Publisher Options..." command

Subscriber to: My doodles
Get Editions:
Automatically
Manually
Get Edition Now
Latest Edition: Friday, February 22, 1991 8:19:27 PM
Cancel Subscriber
Open Publisher
Cancel
OK

Figure 7-5. The standard dialog box for the "Subscriber Options..." command

make any changes to the subscriber, then when the publisher changes, your application should either update the subscriber and throw away any changes the user has made, or update the subscriber and redo all the changes the user has made. When it makes sense, subscribers should be searchable. Subscribers can be cut, copied, and pasted.

The Edition Manager tracks the location of the edition file. If the edition file has been renamed or moved to another location on the same volume, the Edition Manager will be able to find it. However, editions cannot be

moved from one volume to another. To move an edition to another volume, a user must cancel the original publisher, reselect the section, and republish the section on the new volume.

Users can also subscribe to a complete file. In this case, there is no publisher. Your application should check whether the edition file has changed when the subscribing document is opened. Subscribing to files is described in detail later in this chapter.

You've just looked at how your application should implement these new services. Now let's look at some of the details of the recommended user interface for the Edition Manager, and at other aspects of using the Edition Manager.

▶ Displaying Publishers and Subscribers

Apple has recommendations for showing the borders of publishers and subscribers. The technical term for both publishers and subscribers is *section*. Do not use the word *section* in documentation for users; instead use the word *edition*.

The interface design was an interesting challenge for Apple, since sections are basically selections in a document. Unlike normal selections, sections can contain one or more publishers and one or more subscribers. Applications support only one selection. Another difference is that sections persist, whereas selections may or may not persist, depending on the application. "Persist" means that after opening a document, creating a publisher (for example), saving the document, and then reopening the document, the publisher is still in the document and it has the same contents as before.

Publishers and subscribers, when selected, should be highlighted by a 3-pixel line drawn just outside the content of the publisher or subscriber. For publishers, this line should be drawn with a 50 percent gray pattern. For subscribers, use a 75 percent gray line.

Your application can also support an optional command, "Show/Hide Borders," which allows users to see the borders of all publishers and subscribers in a document, or to hide them all. With the "Hide Borders" command, you should show the border of a section only when that publisher or subscriber is selected.

▶ Changes to the Edit Menu

Three new commands should be added to the Edit menu if your application supports the Edition Manager (See Figure 7-1): "Create Publisher...," "Subscribe to...," and "Publisher/Subscriber Options...". The name of the

third command depends on whether a publisher or subscriber is currently selected; it should be grayed out if no section is selected.

Note that these three commands are below cut, copy, and paste and are separated from them by a gray line. Other commands can follow the Edition Manager commands, and they should also be separated by a gray line.

Two optional commands, if supported, should come just after the "Publisher/Subscriber Options..." command. First is the "Show/Hide Borders" command, whose name depends on the current choice of the user. The default state should be "Show Borders"—in other words, section borders should be hidden unless selected. Second is the "Stop All Editions" command, which temporarily stops all update activities for both publishers and subscribers. When the user has selected this command, place a check mark next to it.

If your application will run under both System 6 and System 7, then your application should disable the Edition Manager commands when running under System 6. It should also display publishers and subscribers. Last, it should allow users to edit both publishers and subscribers; otherwise, users will not be able to do anything with them.

▶ Supporting the Edition Manager from Various Types of Applications

Apple recommends how the Edition Manager should be used from four types of applications: word processors, paint programs, drawing programs, and spreadsheets. If your application isn't in one of these categories, use these recommendations as examples of what to do.

Word processors will primarily be subscribers. When publishing from a word processor, the borders around a publisher should move with the text. If the user types into the publisher, then the user will expect to see the borders grow. If the user deletes some text, the borders should shrink.

Paint programs will primarily be publishers. The borders of a publisher will be static, and they will change only if the user explicitly moves them. If your paint program does support subscribers, then it will become more like a drawing program, since a paint program should not allow users to change a subscription. Paint programs should also support the "Show/Hide Borders" command.

Drawing programs will be both publishers and subscribers, and they should support the "Show/Hide Borders" command.

Spreadsheets will primarily be publishers. Borders of publishers will grow and shrink as cells are added or removed from the subscriber. Subscriptions will be used to import other kinds of data, such as pictures.

▶ The Edition File

The edition file contains the latest version of the data from the publisher. If this file is located on an AppleShare server (whether using AppleShare itself or File Sharing), then editions can be used across networks. This enables work groups to easily share data—an increasingly important use of networks. Future versions of the Edition Manager may support other ways of storing editions than just edition files.

An edition file contains data in the same formats as the Clipboard: TEXT and PICT. Every application should be able to subscribe to data in either TEXT or PICT format. Every application should be able to publish data in either TEXT or PICT format. You can support private formats in addition to these two standard formats.

Three data sets are stored in an edition file, identified by the following types: 'alis', 'prvw', and 'fmts'. The data stored as type 'alis' is written by the Edition Manager, contains an alias record for the publishing document, and contains the identity of the section in the document. Since a document can have more than one publisher, the section identity is necessary.

The data stored as type 'prvw' is an optional PICT file that, if present, is displayed in the preview area of the standard publish and subscribe dialogs. Apple recommends that the publishing application create this PICT if the data takes awhile to display.

The data stored as type 'fmts' is a list of the data formats available in this edition file. It also contains the length of the data for each format. This information is maintained by the Edition Manager.

▶ Saving Documents That Contain Sections

When your application saves a document that contains one or more sections, you have a little more work to do. Save each section record as a 'sect' resource in the document's resource fork. Also, save each corresponding alias record as an 'alis' record with the same resource ID.

▶ Using the Edition Manager

In this section, you will look at the calls your application should make to use the Edition Manager. You'll first see how to verify that the Edition Manager is present, and then how to initialize it. Next, you'll look at the most important data structure associated with the Edition Manager and the section record, and at the routines used when working with them. You'll go through the routines to use when working first with publishers and then with subscribers, and look at the routines for subscribing to files.

Last, you'll look at the high-level events your application must support to work with the Edition Manager.

▶ Compatibility and the Edition Manager

Call **Gestalt** with a selector of gestaltEditionMgrAttr to inquire about the Edition Manager. The only attribute that can be returned from this call under System 7 tells you whether the Edition Manager is present. Don't use any of the Edition Manager routines unless you've first verified that it is present.

▶ Starting the Edition Manager

Call **InitEditionPack** to install the Edition Manager in memory and initialize it. This call will fail if there isn't enough memory to load the package. If you cannot install the Edition Manager, then your application should either quit or disable the commands associated with the Event Manager.

▶ The Section Record

Each publisher or subscriber in a document in an open document must be described by a *section record*. The section record contains such information as whether the section is a publisher or subscriber, whether it is to be updated automatically or manually, the time and date it was last changed, a handle to the alias record that points to the publishing document, and a section ID that uniquely identifies this section within this document. It also contains some data that is private to the Edition Manager. For more information on alias records, refer to Chapter 18.

▶ Working with Section Records

As mentioned above, your application must have a section record for each publisher or subscriber in an open document. You can use five routines for working with section records. Use these routines when working with either publishers or subscribers.

Call **NewSection** to create a new section. You must pass the identity of the edition to publish or subscribe to, the location of the document containing the section, the section ID, the update mode (automatic or manual), and whether this section will be a publisher or subscriber. If this call succeeds, it returns the handle to a section record (which contains a

handle to the required alias record). If successful, this routine also calls **RegisterSection**.

Call **RegisterSection** to tell the Edition Manager to keep track of a section. You'll pass the location of the document containing the section and a handle to the section record. This call returns a Boolean telling your application if the edition file was either renamed or moved.

Call **UnRegisterSection** to tell the Edition Manager to dispose of a section. This happens when closing the document containing the section or when the user has canceled the section. The only parameter is the handle to the section record. If this call is successful, you can then dispose of the section record and its alias record. Once a section has been unregistered, you will no longer be notified if it was changed, nor can you read data from it or write data to it.

Call **IsRegisteredSection** when your application is notified that a section has changed. This call verifies that the notification is for a registered section. This is important because your application may have just unregistered a section and received notification that it has changed.

Call **AssociateSection** to ask the Edition Manager to update the alias record of a section when a section record has either been pasted into a new document or been saved as a new document. This call ensures that the alias record will point to the new file.

System 7 has two routines for reading and writing edition data. The data in an edition can be provided in several different formats, and each type of data can be read or written separately. The Edition Manager provides an I/O mark for each type of data. Call **SetEditionFormatMark** to set the mark to an offset into the edition for the specified type of data. Call **GetEditionFormatMark** to find out the value of the mark for the specified type of data.

▶ Creating a Publisher

Call **CreateEditionContainerFile** to create a new (empty) edition file. You'll pass the file specifications for the new file and its creator type. This call doesn't create any data in any format in this new file. The four file types for edition files are 'edtu' for files with no data (as is the case here), 'edtP' for editions with PICT data as their primary data format, 'edtT' for editions with text data as their primary data format, and 'edts' for an edition with any other type of data as their primary data format. You therefore need to provide as many as four different icons for edition files.

Call **DeleteEditionContainerFile** to delete an edition file after the user has canceled a publisher. To provide an undo capability, make this call

after the user has saved the document. If there were any subscribers to this edition, they will receive an error when trying to read data from it.

To get the last edition for which a section was created, call **GetLast-EditionContainerUsed** before creating a new publisher. This routine makes it easy for the user to publish an edition and then subscribe to it. Pass the data returned by this routine to **NewPublisherDialog**, which scrolls the file list to the file described by that data.

Call **NewPublisherDialog** to present the standard new publisher dialog to the user. You'll pass a handle to the "preview" data (if any) and its type. This routine returns a file specification for the new edition file. If your application needs a customized version of this dialog, call **New-PublisherExpDialog**. You must specify five additional parameters to add other items to the dialog.

Call **OpenNewEdition** to prepare to write data to an edition. You need to pass a handle to the publisher's section record, the creator type for the file, and a file specification for the edition file. You'll be returned a refNum if the call is successful.

To write data to an edition, call **WriteEdition** providing a refNum, the data format, a pointer to the data, and the length of the data.

When you are finished writing data to a publisher, call **CloseEdition**. The Edition Manager sets the file type to correspond to the first type of data written to the file. It also sets the modification date field of the section record. Last, it notifies any current subscribers that the edition has been updated.

Call **SectionOptionsDialog** to display the standard Publisher Options dialog box. You'll get an action code back from this dialog indicating the user's choice. An action code of 'writ' means the user selected Send Edition Now; an action code of 'cncl' means the user selected Cancel Publisher; and an action code of '   ' (four space characters) means the user pressed the OK button. This routine also returns a Boolean indicating whether the user has changed the section record, for example, by changing the update mode. To provide a customized version of this dialog, call **SectionOptionsExpDialog**. You can specify five additional parameters to add other items to this dialog.

▶ Creating a Subscriber

To get the last edition for which a section was created, call **GetLast-EditionContainerUsed** before creating a new subscriber. This routine makes it easy for the user to publish an edition and then subscribe to it. Pass the data returned by this routine to **NewSubscriberDialog**, which scrolls the file list to the file described by that data.

Call **NewSubscriberDialog** to present the standard new subscriber dialog to the user. This routine returns a file specification for the selected edition file. If your application needs a customized version of this dialog, call **NewSubscriberExpDialog**. You must specify five additional parameters to add other items to the dialog.

Call **OpenEdition** to prepare to read data from an edition. You'll specify a handle to the subscriber's section record and receive, if successful, a refNum to be used with other calls.

Call **EditionHasFormat** to find out whether the specified edition contains data in the requested format. If the data is in that format, you'll also get the length of that data. To get a complete list of all the data formats available, you can read in the data of type 'fmts'. Remember that this data is simply a list of all the data formats in that edition and their lengths.

To read in data from the edition in a specific format, call **ReadEdition**. In addition to specifying the refNum of the edition and the data format, you also need to provide a pointer to a buffer. The Edition Manager will read in as much data as possible and return a byte count.

When you are finished reading data from a publisher, call **Close-Edition**. The Edition Manager updates the modification date field of the section record.

Call **GetEditionInfo** to get various information about a section's edition file. You'll get the edition's creation date, date of last modification, creator type, file type, and location.

Call **GotoPublisherSection** to open the document containing the publisher of the specified edition. This call sends the Finder one Apple event to open the edition and another Apple event to scroll to the location of the publisher in that document.

Call **SectionOptionsDialog** to display the standard Subscriber Options dialog box. You'll get an action code back from this dialog indicating the user's choice. An action code of 'read' means the user selected Get Edition Now; an action code of 'cncl' means the user selected Cancel Subscriber; and an action code of ' ' means the user pressed the OK button. This routine also returns a Boolean indicating whether the user has changed the section record, for example, by changing the update mode. To provide a customized version of this dialog, call **SectionOptionsExpDialog**. You can specify five additional parameters to add other items to this dialog.

▶ Format Marks

The Edition Manager brings the concept of format marks. A format mark is similar to the current file position maintained by the File Manager, except that there is a format mark for each data format in an edition file.

The format mark indicates where the next I/O operation should resume. Format marks are initially 0, but are set to the next byte following an I/O operation. This allows for reading and writing edition data in chunks, rather than requiring that all the data be handled at one time.

Set the format mark by calling **SetEditionFormatMark**, specifying which Edition file, the data format, and offset. Get the format mark for a particular format by calling **GetEditionFormatMark**, similarly specifying the Edition file and the data format.

▶ Subscribing to Files Rather Than Editions

The Edition Manager also enables users to subscribe to a file. The Edition Manager never does file I/O directly; it does I/O through an edition opener procedure. It uses a standard opener procedure for edition files. These opener procedures enable the Edition Manager to track changes in edition files.

You can write custom opener procs to allow your application to subscribe to entire files. The details for writing an opener proc are spelled out in detail in the Edition Manager chapter of *Inside Macintosh*, Volume VI. Your application must also call several procedures to use an opener proc.

Call **SetEditionOpenerProc** to set the current edition opener proc. You'll pass the address of a opener proc. Call **GetEditionOpenerProc** to get the address of the current edition opener proc.

Call **CallEditionOpenerProc** to call the current edition opener proc, passing a file specification to the file, a section record, and other parameters. You'll also pass an edition opener verb: eoOpen, eoClose, eoOpenNew, eoCloseNew, or eoCanSubscribe. The Edition Manager uses these verbs to describe what actions to take with the file.

Tell the Edition Manager which format I/O procedure to use by calling **CallFormatIOProc**. Specify whether you want to read, write, create, or verify the existence of data in the specified format. Use this call when either subscribing to files or if you need to override the default Edition Manager I/O procedures. You can also create your own I/O procedures for use by the Edition Manager.

▶ Apple Events and the Edition Manager

Your application must be Apple event-aware to use the services of the Edition Manager. In particular, your application must support the 'open' Apple event. This message is sent to publishers when the user requests to go to the publisher of an edition. Actually, the subscribing application sends the message by calling **GotoPublisherSection**.

In addition to supporting Apple events, your application must also deal with the class of high-level events defined for the Event Manager. This class is known as type 'sect'. Only four events are in this class: 'read', 'writ', 'cncl', and 'scrl'.

Your application will receive a 'read' event when a publisher has recently updated an edition that an open document has registered with the Edition Manager. When you get a 'read' event, your application should read the section in again and update the document.

Your application will receive a 'writ' event when it tries to register a section, but the corresponding edition file is missing.

Your application will receive a 'cncl' event when a publisher has canceled an edition. Your application should also cancel the corresponding section in the open document.

Your application will receive a 'scrl' event when the user has selected Open Publisher from the Subscriber Options dialog. Your application will be launched if it isn't open. The 'scrl' event will then be sent to ask your application to scroll to display the publisher.

When it receives any of these four 'sect' events, your application must call **IsRegisteredSection**. This call will check that the section specified in the event record is registered with the Edition Manager. It is possible, as mentioned previously, that the section may have just been unregistered. If the event is for an unregistered section, then your application should ignore it.

▶ Implementing Edition Manager Support

When you are implementing support for the Edition Manager in your applications, follow the human interface guidelines for it carefully. Apple has devoted a significant amount of resources to design and test the interface for the Edition Manager.

Don't use the Edition Manager for purposes for which it isn't suited. For example, the Edition Manager is not well-suited for real-time transfer of data. Use Apple events, high-level events, or the PPC Toolbox to accomplish this instead.

▶ Conclusion

In this chapter, you've looked at the Edition Manager, which provides high-level services for transferring data from one application to others and automatically keeping this data up to date. The Edition Manager can markedly reduce the drudgery of maintaining data in complex docu-

ments. Apple is strongly encouraging all developers to support this new operating-system feature, so you can expect pressure from users for this as well.

Get Info ▶

For more information on the Edition Manager, refer to the Edition Manager chapter of *Inside Macintosh,* Volume VI. You should also refer to the Event Manager and AppleEvent Manager chapters of *Inside Macintosh,* Volume VI for information on high-level events and AppleEvents.

8 ▶ Fonts and TrueType

▶ Introduction

Apple announced a new font technology at the 1989 Worldwide Developer's Conference. This technology had a code name of Royal and has since been renamed TrueType. Quite a controversy arose, since PostScript fonts had been the primary high-quality fonts available on the Macintosh.

In this chapter, you'll look at how fonts have evolved on the Macintosh. To understand TrueType—Apple's new font technology—and its significance, you must first understand the context in which fonts are used. To do this, you'll first look at bitmap fonts and then at PostScript and PostScript fonts. Then you'll look at why Apple invented TrueType.

After all that history, you'll look at what TrueType fonts are and how they differ from PostScript fonts. Last, you'll look at some new routines in the Font Manager that are useful in working with fonts—TrueType or not.

▶ Macintosh Fonts: Then and Now

To understand the significance of TrueType, you'll need to understand the history of Macintosh fonts. As you'll see, both the definition and function of fonts in the Macintosh operating system have evolved from the early days of the Macintosh.

Important ▶ The word *font* changes meaning from section to section. For example, in the next section, it means a 'FONT' resource. Rather than saying a 'FONT' font, I've used the word *font*. Only when the meaning is not clear have I used the more complete description.

▶ The First Fonts: 'FONT' Resources

The first Macintosh, introduced in 1984, defines a font as a complete set of characters of one typeface, with no stylistic variations (bold, italic, and so on). A font can have a maximum of 255 characters; each font must have a "missing" character displayed for any character not available in that font. Each size of a font is stored as a 'FONT' resource, where font size is measured in points. In this version, most of the common Macintosh fonts are supplied in the point sizes 9, 10, 12, 14, 18, 20, and 24. This uses a lot of disk space, since each character in a font is a bitmap.

The size of a 'FONT' font has to be in the range of 1 to 127 points. On the Macintosh, a point is 1/72 of an inch; in typography, a point is 1/72.27 of an inch—not much of a difference, but in high-quality typography it does make a difference. The nominal resolution of most Macintosh screens is 72 dots per inch (both horizontally and vertically), so measuring fonts using the former units is convenient for everyone (except typographers).

'FONT' resources contain several sets of data. First, each font has a series of flags and global parameters. The flags tell whether the font is proportional or fixed width, black or color, and so on. The globals specify ascent, descent, and leading for the font, the maximum kerning, and many other parameters. Following this are three tables: a table of the bitmaps for the characters, a table of offsets into the bitmap table for quickly indexing into the bitmap table, and an (optional) character offset and width table.

'FONT' resources are identified primarily by font number, although almost all fonts also have a name for the convenience of users. Under Apple's plan for font numbers, 0 through 127 are reserved for use by Apple; 128 through 383 are reserved for font vendors to be assigned by Apple; and 384 through 511 are available for use by anyone. There aren't 32,768 font numbers because the 'FONT' resource IDs have composed 9 bits for the font number and 7 bits for the font size.

Unfortunately, this scheme has never worked because it assumes only a small number of fonts on the Macintosh. As it's turned out, many people have tried their hand at creating fonts (perhaps partially because writing

Macintosh applications is a lot harder than writing MS-DOS or CP/M applications). Soon, hundreds of fonts had become available for the Macintosh, some created by professional type designers, most created by others. Few designers, if any, ever bothered to register their fonts with Apple.

Even if everyone had registered, there still would have been problems. Fonts could be used only after the user installed them into his or her System file. A difficult-to-use utility, known as the Font/DA Mover, was used for this purpose. When it installed a font into the System file, it would renumber the font's resource ID if another font in the System already had that ID. As a result, users rarely had the same resource ID for the same fonts. This caused tremendous headaches for people who shared documents because the fonts used in creating the document, which had certain IDs, usually had different IDs on the other person's machine.

The primary output device when the Macintosh 128 and 512 were introduced was the ImageWriter. The Font Manager (through several layers of system software) would send the bitmaps corresponding to the characters from a font to this dot-matrix printer. If the font was not available in the requested size, the Font Manager would attempt to scale the font in another size to the requested size. This usually led to unattractive characters, except when the font was available in an integral multiple or divisor of the requested size.

▶ PostScript Arrives

PostScript and PostScript fonts were first usable on the Macintosh when the LaserWriter was introduced. This 300 dpi laser printer had a PostScript interpreter built into it. At the time, Apple advertised the LaserWriter as its most powerful computer because the Motorola CPU chip in the printer used a higher clock speed than did any members of the Macintosh family.

PostScript fonts differ significantly from bitmap fonts. PostScript fonts do not contain bitmaps; rather, they contain mathematical descriptions of the curves that describe each character in the font. PostScript fonts have several advantages over bitmap fonts:

- A PostScript font can be used for a wide range of point sizes, and it looks good at any size. (Actually, some fonts are designed to look their best only in certain sizes.)
- A PostScript font requires much less space than the equivalent set of bitmap fonts.

- PostScript fonts are referenced by name only, reducing the potential for conflicts between fonts when documents are shared across machines. As you will see later, using names does not eliminate conflicts.

One problem with PostScript fonts is that they cannot be displayed on the screen by the Macintosh system software, because there is no PostScript interpreter built into the Macintosh operating system. What is displayed on the screen is a bitmap font that corresponds to the PostScript font. This correspondence is maintained through the bitmap font name and the name for the equivalent PostScript font. Resource IDs or font numbers are not used for this purpose.

The printer driver (for example, for the LaserWriter) checks whether the desired PostScript font is in the LaserWriter. If the font is not available in the printer, then the driver checks whether the font is available on the user's machine. If not, then at the user's choice, either another PostScript font is substituted or the bitmap font is downloaded and printed. If the bitmap font is printed, the print quality is usually lower, but sometimes this is still the better choice.

When the printer (such as the Apple LaserWriter) needs to print a character in a PostScript font, it generates a bitmap for that character in the requested size and style in real time. Higher-quality PostScript fonts are actually a set of PostScript fonts, one font per style: plain, italic, bold, and bold italic. The character bitmaps generated in the printer are created for the resolution of the printer—300 dpi for the Apple LaserWriter, or 1270 dpi or higher for typesetting devices. Ultimately, even PostScript fonts become bitmaps.

Adobe later released a product called Adobe Type Manager (ATM) that contains a PostScript interpreter. ATM, released after TrueType was announced, traps all references to fonts and checks for a corresponding PostScript font file. If such a file exists, then the interpreter generates bitmaps at the specified size for the specified font. By using PostScript fonts directly on the Macintosh, ATM allows PostScript fonts in arbitrary sizes to be displayed on the screen. An important disadvantage of ATM is that PostScript font files, although more compact than bitmap fonts, still take up a lot of room, especially when a dozen or more fonts are kept on disk.

So what is a PostScript font anyway? A PostScript font contains, for each character in the font, a set of PostScript commands that describe the shape of the character.

▶ New Font Resources: 'NFNT' Resources

After all the problems with the 'FONT' style of fonts, Apple created a new type of bitmap font that was kept in 'NFNT' resources. 'NFNT' resources are identical to 'FONT' resources, except for one difference: 'NFNT' fonts can be in color or gray-scale. A resource of type 'fctb' specifies a color to be used with the 'NFNT' resource, and the 'fctb' must have the same resource ID as the 'NFNT'.

Both old 'FONT' resources and new 'NFNT' resources are organized by 'FOND' resources. A 'FOND' resource manages all the fonts in a single font family, thus reducing some of the complexity of having numerous bitmap fonts. A 'FOND' resource contains a list of all the 'NFNT' or 'FONT' resources in a single font family. A font family is the set of all fonts ('NFNT' or 'FONT') associated with a particular font name, such as Courier or Chicago. In some cases, some font vendors use one 'FOND' resource for all the fonts associated with a single font name and a particular style, such as plain, bold, or italic. They do this so that the bitmap fonts can be matched with their corresponding PostScript fonts. PostScript fonts almost always include one PostScript font for each important style. For example, when you buy a PostScript version of the Times Roman font, you actually get four separate fonts: plain Times Roman, bold Times Roman, the italic version, and bold italics. You get these four fonts rather than a single font because the font designers have decided that for the best-quality output, it is better to design fonts for these styles separately. For other combinations of styles, the printer uses one of these four styles as a base and adjusts the fonts algorithmically to produce the desired style.

'FOND' resources also contain much more information about the font. Among this data are global values that apply to the entire font family and optional tables that describe characteristics of each character in each font in the family. Among the global values are flags specifying whether the font is proportional or fixed in width; the ascent, descent, and leading; and the maximum width of a character. There are two optional tables, one containing character widths, the other containing character image heights. Among the other tables in this resource are a list of all the associated font resources and kerning data for character pairs. 'FOND' resource IDs are 16-bit integers, which means that many more fonts can be identified without the resource ID conflicts that occurred with 'FONT' resources.

Technical Note #191, titled *Font Names*, recommends that applications identify the fonts used in documents by name rather than by resource ID or font number. If the font is part of a font family, then you should save the font family name ('FOND') rather than the name of the specific bitmap

font ('NFNT' or 'FONT'). This solves some of the font identification conflicts, but alas, it does not solve this problem completely.

Unfortunately, conflicts in font names can occur. This tends to happen across vendors, since any font vendor will presumably not sell more than one font with the same name. For the most part, such fonts include a corresponding PostScript font that is used for printing. For an example of this name problem, assume two (or more) vendors sell a font named Helvetica. If a document is created using one vendor's Helvetica and then printed on another machine that has another vendor's Helvetica, then the font widths will probably differ between the two versions of Helvetica. This may change the locations where lines of text wrap, and it can change the page boundaries. Clearly, this is a problem for all applications, especially text-intensive applications.

▶ TrueType Emerges

Apple's goals in developing TrueType were to move away from bitmap fonts and their associated problems, but, at the same time, to provide improvements over the PostScript font technology. Another goal was to move towards an open, non-proprietary font format. At the time TrueType was announced, only Adobe and its licensees could create Type 1 PostScript fonts. The TrueType font format is also designed so that existing fonts in other formats (PostScript and others) can be easily converted. Another goal was to gain the cooperation of the type industry. Apple worked with many leading type vendors to ensure that the TrueType language would provide power and flexibility. Judging from the vendors who are supporting this new font technology—including Linotype, Monotype, International Typeface Corporation, Bitstream, AGFA/Compugraphic, URW, and Kingsley/ATF—Apple seems to have succeeded in doing this.

TrueType technology is used for displaying characters on the screen as well as for printing, so fonts must be rendered quickly. In fact, one advantage of TrueType over PostScript is that fonts are rendered more quickly using the former. This is important in providing fast response time. Another advantage of TrueType over ATM and PostScript fonts is that TrueType is built into the operating system. Users do not have to purchase it, as they do with ATM.

TrueType allows a much closer correspondence between the characters seen on the screen and on paper. Because the fonts are created from the same description, text-intensive applications will have an easier time maintaining the correspondence between screen and paper. In addition, because this technology is embedded into the system software, this

technology should also be more stable than the PostScript/bitmap font technology.

TrueType fonts are stored as 'sfnt' resources. These resources, similarly to 'NFNT' resources, are also managed by 'FOND' resources. Entries for an 'sfnt' in a 'FOND' resource are distinguished by a value of 0 for the size field; this value will be nonzero for an 'NFNT'.

These new 'sfnt' resources are considerably improved from the earlier font resources beyond the obvious change from bitmaps to outlines. The 'sfnt' resource has a version number, allowing for future changes in the structure of the resource. The font data has been separated into tables, each table named with a tag, or 4-byte code. Thirteen tables are defined in the first version of the 'sfnt' resource, with provisions for additional tables in the future. Each table also has a version number, allowing for changes. There is also a checksum for the entire resource and for each of the tables. You can use these checksums to detect changes in a font between the time when a document is changed and saved, and later when it is opened.

Several of the more important tables in the 'sfnt' resource are the 'head' table, which contains a font header; the 'glyf' table, which contains the information for generating each character; the 'name' table, which contains the font family name, a copyright notice, a full font name, a version string, and other information, possibly in a variety of languages; and the 'hdmx' table, which contains horizontal metrics for use in laying out text. The 'sfnt' resource is quite complex.

The checksums and the information in the 'name' table enable the font identification problem to be completely solved, so long as TrueType fonts are used. By saving the 'sfnt' checksum with the font name in documents, and by checking this information when the document is opened, your application can warn the user if the font information is different since the document was last saved. A font may have changed because additional kerning values were added, because new ligatures were added, or for related reasons. If the font has changed, then you can warn the user that the appearance of the document may change because one or more fonts have changed.

Sophisticated typographic applications can also take advantage of the 'sfnt' resource to easily locate font metric information. Previously, applications had to directly access font bitmaps and calculate metrics from them. This is tricky code to write, and you will have to continue using such code for non-TrueType fonts.

TrueType fonts, unlike bitmap fonts ('FONT' and 'NFNT'), are outline fonts. They can therefore be used in arbitrary point sizes, with no penalty for using less common sizes. Characters in a TrueType font on the screen correspond closely to the high-quality printed characters. TrueType fonts

are black; that is, TrueType fonts at this time cannot specify colors or grays for individual pixels.

Moreover, unlike PostScript fonts, TrueType fonts are completely integrated into the operating system—they are used by the Macintosh system software to generate characters for both screen and printer. Furthermore, you can use TrueType fonts on any output device, not only on laser printers; PostScript fonts are used only for printing on laser printers, unless you choose to use Adobe's ATM product. Finally, you can use TrueType fonts in the same document as PostScript fonts.

▶ The PostScript Controversy

PostScript fonts come in two formats. The only public font format available when PostScript was released was the Type 3 format, which was widely supported by many type vendors. Adobe had another format, the Type 1 format, which produced better-quality output because it provided for more advanced capabilities, such as providing hints for generating the bitmaps at a particular size. For the first several years, Adobe kept the Type 1 format proprietary. A handful of companies paid a licensing fee to use the Type 1 format and obtain better-quality fonts than the Type 3 equivalent.

Apple and Microsoft announced an agreement at the 1989 Seybold Conference, the premier trade show and conference on electronic publishing and digital typography: to have Apple license its new font technology to Microsoft for use in Windows and Presentation Manager. The market for TrueType fonts is therefore larger than if the technology existed solely on the Macintosh. This will encourage more font vendors to offer TrueType fonts. In reaction to this agreement, several months later Adobe announced that it would make the Type 1 format an open format—that is, it would be documented and available for use by anyone with no licensing fees. In addition, the documentation on the Type 1 format is available from Adobe for a nominal fee.

▶ TrueType Font Technology

Each TrueType character is described by one or more outlines. Each outline is composed of one or more contours. A contour is a second-order B-spline or a straight line, in the simplest case. The B-splines in TrueType use two kinds of points to describe a curve: points on the curve and points off the curve. PostScript uses cubic Bezier equations, which require more calculations than quadratic splines. For some characters, however, more

splines are required to describe a given outline in TrueType than Bezier curves in PostScript.

Figure 8-1 shows a letter from a TrueType font. Compare it with Figure 8-3, which shows a letter from a PostScript font. Notice that there are more points required to specify the TrueType letter than the corresponding PostScript letter. Figures 8-2 and 8-4, which show a closeup of a portion of the same letters in Figures 8-1 and 8-3 respectively, also show this.

The TrueType instruction set is better suited to create fonts than is PostScript's, since TrueType was designed solely and specifically for that purpose. PostScript fonts are written using PostScript commands, but PostScript is a general page-description language, not a language specific to creating fonts.

The TrueType language contains about a hundred instructions. The language itself is stack-based, as is PostScript. The instructions are, to a C or Pascal programmer, an odd mix of high-level, application-specific instructions and low-level instructions, some seeming close to assembly language. The TrueType instructions fall into the following categories:

- Modifying the graphics state settings
  - Controlling the global graphics state
  - Controlling the local graphics state
  - Setting the control points for a contour
  - Controlling how values should be rounded
- Managing outlines
  - Interpolating and shifting contour points
  - Moving contour points
  - Reading and writing data
  - Delta exceptions
- General-purpose instructions
  - Stack manipulation
  - Relational and logical instructions
  - Flow-of-control instructions
  - Arithmetic and mathematical instructions
  - Creating and calling functions
  - Debugging
  - Compensating for engine characteristics

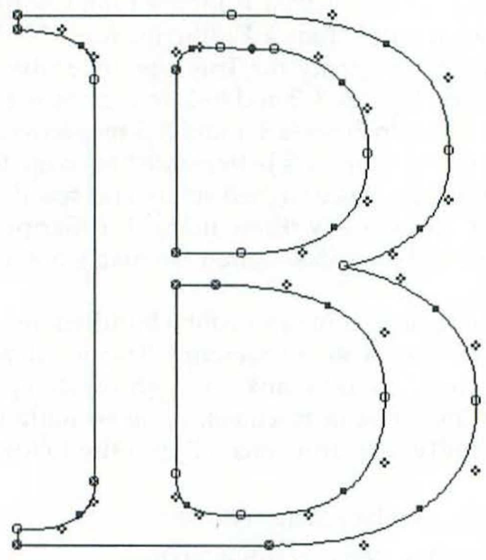


Figure 8-1. A TrueType letter

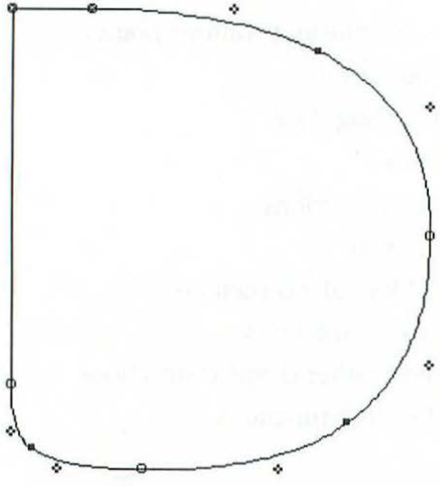


Figure 8-2. Closeup of a TrueType letter

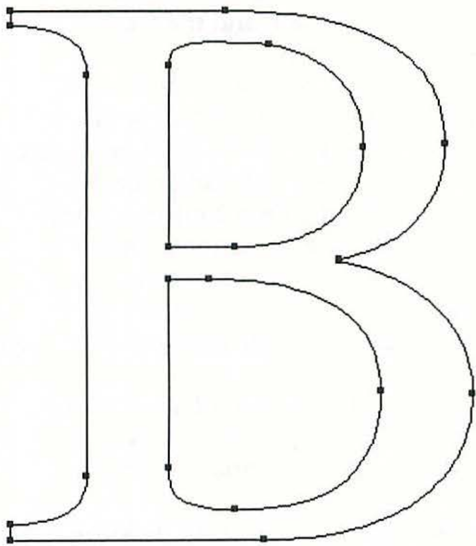


Figure 8-3. A PostScript letter

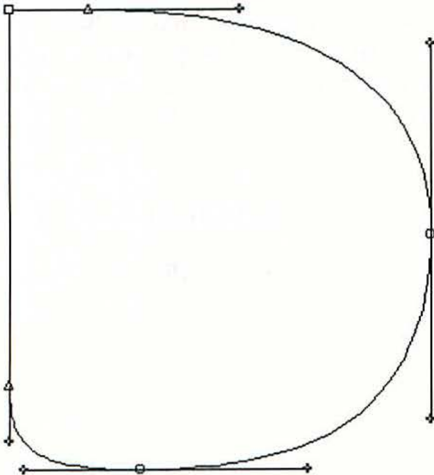


Figure 8-4. Closeup of a PostScript letter

The TrueType language assumes the existence of a stack, a local graphics state, and a global graphics state. The two graphics states contain constants, variables, pointers, and flags. The global graphics state is initialized with each new font and size used, and the local graphics state is initialized when a new character is created.

The language is quite interesting because it is an example of a truly application-specific language. Some features of the language are specific to the creation of fonts, and therefore only knowledgeable type designers will understand some aspects of this language. Because the TrueType language is used solely for fonts, only font-designing applications will use this language. Even in those applications, font designers will almost certainly never see the TrueType language directly.

TrueType outlines are converted into bitmaps by a three-step process. These processes are performed, in turn, by the scaler, interpreter, and scan converter.

First, the *scaler* takes the outline of a character and scales it to the requested point size. The output from this process is another outline, which is measured in units appropriate to the output device.

Note ▶ The original outline is measured in units known as FUnits, or Font Units—an abstract unit of measure that is small compared to the height or width of a character. An FUnit is typically 1/2048 to 1/32768 the width of an em, a standard measure of the width of the letter *M* in the font.

Second, the *interpreter* takes the outline from the scaler and interprets any TrueType instructions for the character. Outline fonts do not necessarily include any instructions. This process is known as *grid-fitting* because the interpreter takes the character outline and fits it onto a device-dependent grid. The output of the interpreter is an outline of a particular character for use on the current output device. As a result of this process, this new outline may be changed or distorted with respect to the original outline, so that the final character will present the best appearance of the font.

Third, the *scan converter* takes the modified outline from the interpreter and generates a bitmap rendition of the particular character for use on the current output device.

Taking Full Advantage of TrueType

If an application knows that it is working with an outline font and not a bitmap font, it can provide new capabilities for working directly with text. This is primarily because outline fonts can be used at arbitrary sizes, unlike bitmap fonts. Applications could allow a user to directly manipulate the size of fonts instead of requiring a menu command to accomplish the same function. Similarly, users could specify horizontal and vertical letter spacing.

Users could also select some text and directly specify constraints instead of, for example, using trial and error to fit a title so that it exactly fills the width of a column. Using TrueType, you could allow the user to select a line of text and then specify the desired constraints interactively, that is, that the text should be scaled so that it was exactly as wide as the column.

System 7 ships with four TrueType font families: Times, Helvetica, Courier, and Symbol. Apple has licensed these fonts from Linotype and the International Typeface Corporation; Adobe previously licensed these fonts to create the PostScript versions of them. Thus, the TrueType fonts will match the metrics of the PostScript fonts in the Apple LaserWriter family.

Using the Font Manager

You will now see how the Font Manager chooses a font, how to check that outline fonts are available, and the new Font Manager routines.

How the Font Manager Chooses a Font

The default behavior of the System 7 version of the Font Manager is as follows when your application requests a specific font and size. The Font Manager first looks for a bitmap font that meets the request. The Font Manager prefers bitmap fonts to outline fonts for reasons of compatibity. Your application can tell the Font Manager that it would prefer outline fonts by using the new **SetOutlinePreferred** system call described below.

If the Font Manager cannot find a bitmap font that matches your request, it looks for an 'sfnt' resource that will match the request. If it can find an 'sfnt', it will use that for the font. If it cannot find an 'sfnt', the Font Manager will use the algorithm described in the Font Manager chapter of *Inside Macintosh*, Volume I to scale another bitmap font to fulfill the request.

▶ Compatibility and the Font Manager

Call **Gestalt** with a selector of gestaltFontMgrAttr to inquire about the Font Manager. The only attribute this call can return under System 7 tells you whether outline fonts are available.

▶ Using the New Font Manager Calls

The Font Manager in System 7 contains several new routines for working with outline (TrueType) fonts. These calls are not available under older versions of the system. Most applications will not need these calls, but text-intensive applications, especially high-end applications, will use these calls to provide the highest-quality output. Note, however, that even with TrueType, the Font Manager does not provide any support for rotating text, skewing text, or kerning. If you require any of these capabilities, you'll have to write the code yourself.

Call **SetOutlinePreferred** to tell the Font Manager that your application would prefer an outline font to a bitmap font if both are available. The two fonts must be identical in name, style, and point size. This call affects all open grafPorts of the calling application. By default, this global value (**OutlinePreferred**) is FALSE for compatibility with older applications. Call **GetOutlinePreferred** to find out the current state of **OutlinePreferred**.

To find out whether the current font in the current grafPort is an outline font or a bitmap font, call **IsOutline**. You must specify the point size you're interested in when making this call.

Call **FlushFonts** to flush the Font Manager's caches, to get the Font Manager and its caches into a known state. The Font Manager saves the most recently used bitmaps of characters and other font data so that it can provide good performance. By flushing these caches, subsequent references to outline fonts will require regenerating the character bitmaps. Font-editing programs are about the only applications that would use this call.

To get accurate measurements on a string of characters, Call **OutlineMetrics**. You pass a string of text and the point size that you are interested in. The call returns such information as the following:

- Height of the tallest character in the string.
- Height of the lowest descender of characters in the string.
- An array of advance-width measurements, one per character in the string. The *advance-width* measurement of a character is the width of the character plus the width of the white space on both sides of the character.
- An array of left-side bearing measurements, one per character in the string. The *left-side bearing* measurement of a character is the width of the white space to the left of the character.
- An array of bounding boxes, one per character in the string. Each box is the smallest box that will fit around the nonwhite pixels of that character.

OutlineMetrics provides a standard routine for getting font measurements without having to manipulate the bitmaps of characters, which was the only way to accomplish this task previously. Unfortunately, this routine works only with TrueType fonts.

By making a call to **OutlineMetrics** before calling **DrawText** (the usual routine to draw a string of text), you can tell whether any character in the string will exceed the line spacing.

If a character will exceed the line spacing, then that character may overlap a character on another line. You have a choice: You can adjust the line spacing so that that character will not overlap characters on other lines, adjust the line spacing for the paragraph, allow the characters to overlap, and so on. Calling **SetPreserveGlyph** with a value of TRUE forces the Font Manager to draw the character exactly as described by the font, but not to scale the font to fit the line. Otherwise, the Font Manager will scale the character so that it will not overlap characters on other lines. By default, the state of this global value (**PreserveGlyph**) is FALSE for compatibility. **PreserveGlyph** affects all grafPorts in the current application. Call **GetPreserveGlyph** to retrieve the current value.

Last, by calling the existing routine, **RealFont**, with a specified font and size, you can find out whether an outline font can be used at that size. Size is a characteristic of the font and is therefore decided by the font designer. Some fonts may not be usable at very small or large sizes by the designer's decision.

▶ Conclusion

In this chapter, you've looked at the history of fonts in the Macintosh operating system. Fonts started as bitmaps, then higher-quality PostScript fonts were supported by the Print Manager. With System 7, Apple has introduced the latest font technology: TrueType. This technology uses one description for the font as outlines for both screen and output devices. Apple has not made any of the older font technologies obsolete. 'FONT', 'NFNT', and PostScript fonts will all continue to work under System 7, so any existing investment in fonts can continue to be used.

For the most part, applications are not affected by these changes in font technology. Nonetheless, users will see significant improvements on the screen because arbitrary font sizes are now available, and because the same fonts will be used on the screen and on output devices. Several new calls to the Font Manager are of interest primarily to text-intensive Macintosh applications.

Get Info ▶

For more information on the Font Manager and the new calls discussed in this chapter, refer to the Font Manager chapter of *Inside Macintosh,* Volume VI. For more information on the TrueType instruction set and the format of the 'sfnt' resource, refer to *The TrueType Font Format Specification,* available from APDA. For a comparison between PostScript and TrueType fonts, read the article entitled "Font Wars" by L. Brett Glass in the August 1990 issue of *Byte.*

Several books can provide you with more information on PostScript fonts. First, three PostScript reference books deal with various aspects of fonts. The *PostScript Language Reference Manual,* Second Edition (Addison-Wesley, 1990) and the *PostScript Language Tutorial and Cookbook* (Addison-Wesley, 1985) do not deal exclusively with fonts, but nonetheless do have important information about them. The *PostScript Language Reference Manual* describes the Type 3 font format. The *Adobe Type 1 Font Format* (Addison-Wesley, 1990) describes the Type 1 font format in great detail.

Two other useful books on PostScript programming provide examples and more information about PostScript fonts. These are *Understanding PostScript Programming,* by David Holzgang (Sybex, 1987) and *Real World PostScript,* edited by Stephen Roth (Addison-Wesley, 1988).

9 ▶ TextEdit and International Services

▶ Introduction

In this chapter, you'll look at improvements to TextEdit, the operating-system component that provides fundamental text-editing services. As you will see, most of these changes prepare TextEdit for working with the Script Manager.

After this, you will look at improvements to the Script Manager itself. This manager provides a sophisticated set of routines that can make any application work with most of the writing systems of the world. You'll then look at the international resources and services that enable your application to be easily translated for use in the other languages of the world.

Last, you'll look at how to design international support into your application. By designing it into the first version, you can save yourself a lot of work in the future.

Before looking at TextEdit, the Script Manager, and the International Utilities Package, however, let's discuss why Apple is providing these services.

▶ Why the Script Manager?

Apple is devoting a lot of resources to ensure that the Macintosh system software can be used in many of the world's languages using the appropriate writing system. Why is Apple spending so much time, money, and effort to do this when most other manufacturers don't seem to be interested?

The answer comes when you look at Apple's revenues: It makes good business sense. More than 40 percent of its revenues now come from sales

outside the U.S. This is happening for several reasons. The computer market in the U.S. isn't saturated yet, but this country has more computers per person than any other country in the world. The markets in Europe, Asia, and Australia are growing at a faster rate than in North America. The computer markets in Latin America and Africa are starting to grow more quickly as well.

Most other operating systems provide little, if any, support for other languages. MS-DOS and most of the operating systems running on minicomputers and mainframes fall into this category. The UNIX operating system is in the process of being internationalized, but it is still a long way from providing the kind of support that the Macintosh operating system does. Many versions of UNIX sold today still support only 7-bit character codes, which gives you some idea of how far they have to go. Basically, most operating systems work best with English (or another European language) because most American engineers have little knowledge of or exposure to other languages or writing systems. Support for other languages adds little additional cost or development time if it is supported in the initial design. Retrofitting it into existing products is considerably more difficult. Undoubtedly other operating systems will someday be enhanced to support other languages, but the Macintosh operating system is far ahead of them.

The Macintosh system software uses the Script Manager and the International Utilities Package to support the language of each country, and also supports the appropriate currency symbols, calendars, and keyboards. Apple is thereby ensuring that no technical barriers will hinder its participation in the rapid growth of the computer markets in other countries. Apple Japan is gearing up to do more than a billion dollars' worth of business in the next several years.

If your application uses the Script Manager and the International Utilities Package, then your application can also participate in these rapidly growing markets. Once again, if you add these features to your initial design, it will add little to the development time or costs. Retrofitting an existing product can take much more time and money. Text-intensive applications are affected more than other kinds of applications.

▶ Improvements in TextEdit

TextEdit provides the basic text-editing services for the Macintosh operating system. TextEdit is used throughout the operating system as well as by all applications. Before looking at the improvements to TextEdit in System 7, let's review what TextEdit should and shouldn't be used for.

TextEdit is not a word processor—it can handle no more than 32K characters, and its performance declines well before that limit is reached. Although the latest version supports not only styled text, but also multiple script systems, TextEdit does not support tabs or margins. TextEdit was intended for use in dialog boxes and other places where a limited set of text-editing services were needed.

TextEdit has fewer routines than QuickDraw, the basic graphics package of the operating system. However, text editing is more complex than graphics—just look at the data structures involved as described in *Inside Macintosh*. Fortunately, most applications do not have to worry about the details. Those that do can take advantage of some customization hooks put into TextEdit.

▶ Compatibility and TextEdit

Call **Gestalt** with a selector of gestaltTextEditVersion to get the version of TextEdit. Five values are defined:

- gestaltTE1 (= 1) for TextEdit in the Mac IIci ROM
- gestaltTE2 (= 2) for TextEdit with 6.0.4 Script Systems on the Mac IIci (which fixes Script Manager bugs in the Mac IIci)
- gestaltTE3 (= 3) for TextEdit with 6.0.4 Script Systems on all machines except the Mac IIci
- gestaltTE4 (= 4) for TextEdit in System 6.0.5 and later
- gestaltTE5 (= 5) for TextEdit in 7.0

Note that if the current system is using an older version of TextEdit, this call will fail.

TextEdit is compatible with the Script Manager starting with version 2. The **TEFeatureFlag** routine is available starting with version 4.

▶ Script Manager Support in TextEdit

The biggest change in the System 7 version of TextEdit is that it now fully supports the Script Manager. This is good news for anyone developing an application that will be internationalized, because TextEdit will handle more of the low-level work.

Under System 7, TextEdit now supports text from more than one script system and correctly deals with scripts having different primary line directions. That is, TextEdit now fully supports scripts that are written right-to-left (for example, Hebrew and Arabic), as well as those written

left-to-right (for example, English, Spanish, French, and so on). Previously, TextEdit supported multiple styles of text but not scripts. TextEdit also now correctly highlights text written with scripts having different primary line directions, and it properly handles the ambiguous case when the user clicks the mouse between two characters of different line directions. In this case, TextEdit displays a split caret, where the high caret (the top portion) marks the primary caret position for the character offset in the primary line direction. The low caret marks the secondary caret position for the character offset in the secondary line direction. Last, TextEdit correctly handles the movement of the cursor as the user presses the arrow keys to move through text of mixed directions.

TextEdit also maintains synchronization between the keyboard and the display. The user can type on a keyboard using one script and display the characters on the screen using another script, and TextEdit translates the keystrokes into characters to be displayed. If the user clicks the mouse on a character using another script, TextEdit automatically switches the current script to match that of the character.

Finally, TextEdit now supports the double-byte characters used for languages such as Chinese, Japanese, and Korean. It therefore performs the usual action when the user presses the arrow keys or the Backspace key. TextEdit uses the Script Manager to do this and other functions.

▶ Using the New TextEdit Routines

Call the **TEFeatureFlag** routine to control or check the status of features available in TextEdit: outline highlighting, text buffering, whether inline input services are available, and whether they are currently in use. This routine has a unique interface. The parameters for this routine are the feature you are interested in (highlighting or buffering), a handle to the TextEdit record of interest, and an action code. If the action code is TEBitSet, the feature will be enabled, and if the action code is TEBitClear, the feature is disabled. If the action code is TSBitTest, the routine returns the action code corresponding to the status of that feature. If it returns TEBitSet, you know the feature is enabled; if it returns TEBitClear, you know it is not enabled.

Outline highlighting, illustrated in Figure 9-1, provides TextEdit with a behavior similar to MPW. When a window in the background has selected text, an outline is drawn around the selection. This allows the user to see what was selected, but it's also clear that text is not in the front window. Currently, most applications remove all highlighting from selections in background windows.

```
HD:MPW:Sys7Includes:CIncludes:TextEdit.h
struct TEStyleRec {
    short nRuns;                              /*number o
    short nStyles;                            /*size of
    STHandle styleTab;                        /*handle t
    LHHandle lhTab;                           /*handle t
    long teRefCon;                            /*reserved
    NullSTHandle nullStyle;                   /*Handle t
    StyleRun runs[8001];                      /*ARRAY [0
};

typedef struct TEStyleRec TEStyleRec;
MPW Shell
```

```
HD:MPW:Sys7Includes:CIncludes:TextEdit.h
struct TEStyleRec {
    short nRuns;                              /*number o
    short nStyles;                            /*size of
    STHandle styleTab;                        /*handle t
    LHHandle lhTab;                           /*handle t
    long teRefCon;                            /*reserved
    NullSTHandle nullStyle;                   /*Handle t
    StyleRun runs[8001];                      /*ARRAY [0
};

typedef struct TEStyleRec TEStyleRec;
```

Figure 9-1. Outline highlighting

Text buffering is not directly visible to users. Rather, it provides a significant boost to the performance of TextEdit when used with script systems for languages such as Japanese, Chinese, and Korean. When this feature is enabled, TextEdit uses a buffer for storing the input from each call to **TEKey**. You can insert the entire buffer into your TERecord at one time rather than inserting the characters one at a time. A TERecord is the primary data structure used by TextEdit, and is described in detail in *Inside Macintosh*.

You must be careful in using the text buffering feature because the buffer TextEdit uses to store characters is a global, or system-wide, buffer—that is, there is only one such buffer in the entire operating system. Therefore, to avoid having the characters typed by your user in another window, you must call **TEIdle** frequently, especially before any pause longer than a couple of clock ticks. Be careful to avoid turning text buffering off in the middle of a two-byte character. If you are using more

than one TERecord in your application, it's best to wait for an idle event to enable or disable text buffering in the second record.

The inline input services are used with two-byte character codes. These new input services allow double-byte characters to be typed and converted inline instead of requiring the use of a special input window. If your application can use these new services, then you should set **useTextEditServices** in the application's 'SIZE' resource.

Your application can customize the behavior of TextEdit by passing the address of a procedure to the **TECustomHook** routine. The parameters for this routine are the identifier of the TextEdit hook procedure you are replacing, the address of the new procedure, and the handle to the affected TERecord. You can change the following six hook procedures:

- **TEEOLHook**—Determines whether an incoming character is an end-of-line character. If it is, it sets status flags and returns. The default is to compare the incoming character with a carriage return.
- **TEWidthHook**—Measures various portions of a line of text. In practice, TextEdit uses this procedure only when the Roman script system is in use. When a non-Roman script is in use, TextEdit calls the **nTEWidthHook** procedure instead. When the Roman script system is in use and you have defined a new **TEWidthHook** procedure, TextEdit calls your procedure. Otherwise, it calls its **nTEWidthHook** procedure. The default procedure calls **Text-Width**.
- **nTEWidthHook**—Works the same as the **TEWidthHook** procedure. An **nTEWidthHook** procedure can use TextEdit's measuring routine when working with a non-Roman script. You can also customize this routine, but be sure that your new routine works correctly with non-Roman scripts. The default procedure calls **Char2Pixel** or **TextWidth**, depending on the current script system.
- **TEDrawHook**—Draws the components of a line of text. The default procedure calls **DrawText**.
- **TEHitTestHook**—Determines the character closest to the specified position. The default procedure calls **TextWidth** and then **Pixel2-Char** and returns.
- **TEFindWord**—Finds word breaks in a line of text. It replaces the previous hook procedure, **WordBreak**. If you replace the existing **TEFindWord** procedure with your own and your procedure does not handle non-Roman scripts, then call the existing procedure for such cases.

The **nTEWidthHook** and **TEFindWord** hook procedures are new with System 7.

Because TextEdit is now using the Script Manager's **FindWord** procedure, the definition of a word has changed. Previously, TextEdit's own **WordBreak** procedure allowed punctuation marks as part of a word; this is no longer the case except with Roman scripts. Also, TextEdit now selects a run of spaces as a word, whereas previously it would select only a single space.

The **TEKey** routine, the basic TextEdit routine that processes all incoming characters, has been enhanced so that it does not delete a style if the user backspaces to the beginning of that style. If the user does not use this style by clicking somewhere else, or continues deleting characters, then the style is removed.

The **TEGetPoint** routine, which returns the coordinates of the lower left point of the character at the specified offset, has also been enhanced in System 7. It now works when no text is in the TERecord. When the offset is at the end of a line (an ambiguous position), it returns the coordinates for the beginning of the next line.

The **TESetStyle** routine, introduced in *Inside Macintosh*, Volume V, has been enhanced to accept a new mode named doToggle. This mode, when used in conjunction with other modes such as doFace, will remove or add the specified characteristics from the current selection if they are either present or absent across the selection, respectively. For example, if the mode is doToggle + doFace and the entire selection is in italics, then the italic style will be removed from the entire selection.

To find out whether an attribute (font, size, color, and so on) is continous in the current selection of a TEHandle, call **TEContinuousStyle** with the attribute in which you are interested. Call **TENumStyles** to find out the number of style changes contained in a range of text in a TEHandle.

The **TESetJust** routine now has a new set of names for the justification modes that better describe what TextEdit will do with scripts having either primary line direction. Previously, these choices were named teJustLeft, teJustCenter, teJustRight, and teForceLeft. The new names for the justification modes are as follows.

- teFlushRight (with the same effect as the old teJustRight)
- teFlushLeft (previously teForceLeft)
- teCenter (previously teJustCenter)
- teFlushDefault (previously teJustLeft)

The old constants are still defined for compatibility. In the new teFlushLeft mode, justification depends on the direction of the script. For left-to-right scripts, text will be left justified; for right-to-left scripts, text will be right justified.

▶ Improvements in the Script Manager

The Script Manager contains many routines. Some of these routines are implemented by the Script Manager, and others are primarily implemented in a script system. A *script system* is a set of external code segments and resources that support a particular script or writing system. For example, ZhongwenTalk provides support for the Chinese writing system. The Roman Script System supports the writing system used by English and the majority of the European languages. A user can install one or more script systems on his or her Macintosh. However, the Roman script system is built into the operating system and is therefore available on all machines.

Note that the Script Manager is intimately tied to the International Utilities. The Script Manager and the script systems provide support for text services in various languages. The International Utilities support the measurement system (metric or not), number formats, time formats, date formats, and currency formats for a particular country and language.

A given script system might support more than one language. For example, the Roman Script System supports English, French, German, Italian, and most of the other European languages, with the exception of Greek. The Devanagari Script System supports Hindi, Nepali, and Sanskrit. Furthermore, some languages are supported in one or more countries or regions. For example, the English language is used in the U.S., the United Kingdom, Ireland, Canada, Australia, and New Zealand. Each of these countries has a different currency; all of them, with the exception of the U.S., use the metric system of measurements. The names of the months are spelled slightly differently from country to country.

When Apple prepares a version of the system software for a particular country, the internationalized system may include another script system in addition to the Roman script system. It will also include a set of international resources customized for that country. Codes for the various script systems, languages, and regions are listed in Table 9-1. These codes are used by various Script Manager routines and in the International Utilities Package. Note that although codes are defined for most of the world's languages, Apple is not shipping a version of the Macintosh system software for all of them yet.

Table 9-1. Script interface systems, language codes, and region codes

Script Code	*Language Code(s)*	*Region Code(s)*
smRoman	langEnglish	verUS
		verBritain
		verAustralia
		verIreland
	langFrench	verFrance
		verFrCanada
		verFrSwiss
		verFrBelgiumLux
	langGerman	verGermany
		verGrSwiss
	langItalian	verItaly
	langDutch	verNetherlands
	langSwedish	verSweden
	langSpanish	verSpain
	langDanish	verDenmark
	langPortuguese	verPortugal
	langNorwegian	verNorway
	langFinnish	verFinland
	langIcelandic	verIceland
	langMaltese	verMalta
	langTurkish	verTurkey
	langLithuanian	verLithuania
	langEstonian	verEstonia
	langLettish = langLatvian	verLatvia
	langLappish	verLapland
	langFaeroese	verFaeroeIsl
	langGreek	*(Greek in Roman script)*
	langCroatian	verYugoCroatian
	langFlemish	
	langIrish	
	langRomanian	
	langCzech	
	langSlovak	
	langSlovenian	
	langAlbanian	
	langWelsh	
	langBasque	
	langCatalan	
	langIndonesian	
	langTagalog	
	langSomali	
	langSwahili	

Table 9-1. Script interface systems (continued)

Script Code	*Language Code(s)*	*Region Code(s)*
	langRuanda	
	langRundi	
	langChewa	
	langMalagasy	
	langMalayRoman	
	langQuechua	
	langGuarani	
	langAymara	
	langLatin	
	langEsperanto	
	langJavaneseRoman	
	langSundaneseRoman	
smGreek	langGreek	verGreece
		verCyprus
smTradChinese	langTradChinese	verTaiwan
smSimpChinese	langSimpChinese	verChina
smJapanese	langJapanese	verJapan
smKorean	langKorean	verKorea
smArabic	langArabic	verArabic = verArabia
	langUrdu	verPakistan
	langFarsi = langPersian	verIran
	langAzerbaijanAr	
	langPashto	
	langKurdish	
	langUighur	
	langKashmiri	
	langMalayArabic	
smExtArabic	langSindhi	
smHebrew	langHebrew	verIsrael
	langYiddish	
smCyrillic	langRussian	verRussia
	langUkrainian	
	langByelorussian	
	langSerbian	
	langUzbek	
	langKazakh	
	langMacedonian	
	langBulgarian	
	langMoldavian	
	langKirghiz	

Table 9-1. Script interface systems (continued)

Script Code	*Language Code(s)*	*Region Code(s)*
	langTajiki langMongolianCyr langTatar langTurkmen langAzerbaijani	
smSlavic = smEastEurRoman	langPolish langHungarian	verPoland verHungary
smGeorgian	langGeorgian	
smArmenian	langArmenian	
smDevanagari	langHindi langSanskrit langMarathi langNepali	verIndiaHindi
smGurmukhi	langPunjabi	
smGujarati	langGujarati	
smOriya	langOriya	
smBengali	langBengali langAssamese	
smTamil	langTamil	
smTelugu	langTelugu	
smKannada	langKannada	
smMalayalam	langMalayalam	
smSinhalese	langSinhalese	
smBurmese	langBurmese	
smKhmer	langKhmer	
smThai	langThai	verThailand
smLaotian	langLao	
smTibetan	langTibetan langDzongkha	
smMongolian	langMongolian	
smEthiopic = smGeez	langAmharic langTigrinya langGalla = langOromo	
smVietnamese	langVietnamese	
smUninterp		*(Font script is uninterpreted symbols)*
smRSymbol		*(Font script is right-left symbol)*

Table 9-2 lists the codes for the various calendars supported by the Macintosh system software. These codes are also used by various Script Manager routines and in the International Utilities Package.

Table 9-2. Script Manager calendar codes

Calendar Code Name	*Calendar Value*
calGregorian	0
calArabicCivil	1
calArabicLunar	2
calJapanese	3
calJewish	4
calCoptic	5
calPersian	6

Before looking at the routines of the Script Manager, you should know about one important aspect of resources and the Script Manager. Script systems are currently numbered from 0 to 32, although the Script Manager can handle as many as 64 scripts at a time. Resources associated with a script are numbered within the resource ID range of that script. The Roman script system has the largest range of resource IDs, 0 to 16383, because so many languages and fonts are available for them. For all other script systems, the range of resource IDs begins at 16384 + (512 * (scriptCode − 1)). In particular, 'FOND' resources are numbered in the range associated with the script for the language using that font. The Font Manager uses the resource IDs to perform the usual action, such as substituting another font belonging to the same script when the requested font is not available. 'FOND' IDs 0 and 1 are special values for the system and application fonts. Even though they are in the range of the Roman script system, they are not necessarily Roman fonts.

▶ Using the Script Manager

To understand the new routines of the Script Manager, you will have to understand the routines previously available. Notice that most of the routines in the Script Manager work at a lower level than those of the TextEdit routines. This explains why it is difficult to retrofit the Script Manager into existing applications, and why the cost of designing them into the first version of an application isn't high.

Important ▶ Once again, if there is any chance that your software will be used with a non-Roman script, support the Script Manager in version 1 of your application. It will save you a lot of work in the future.

The routines of the Script Manager are listed in Table 9-3 along with a brief description of their purpose. These routines are grouped by the categories defined in *Inside Macintosh*, Volume VI. Routines that are new with System 7 are so marked.

Table 9-3. The Script Manager routines

Category	*Routine Name*	*Purpose*
Working with global and local script variables	GetEnvirons	Gets Script Manager (global) variables
	SetEnvirons	Sets Script Manager (global) variables
	GetScript	Gets script system (local) variables
	SetScript	Sets script system (local) variables
Checking and setting system variables	GetAppFont	Gets the resource ID of the current application 'FOND'
	GetDefFontSize	Gets the default font size
	GetMBarHeight	Gets the height of the menu bar in the current system font
	GetSysFont	Gets the resource ID of the current system 'FOND'
	GetSysJust	Gets the current default line direction
	SetSysJust	Sets the current default line direction
Setting keyboard script	KeyScript	Sets the keyboard or keyboard script
Getting script information	FontScript	Gets the script code for the current font in the current grafPort
	IntlScript	Gets the script code for the current font in the current grafPort for use with the International Utilities Package
	Font2Script	Translates a font ID into a script code
Getting character information	CharByte	Is byte part of a double-byte character?
	CharType	What type of character is this byte?
	ParseTable	Returns a table used for parsing characters as single-byte/double-byte
Drawing and editing text	Char2Pixel	Finds position for caret given a pointer to text in a buffer
	NChar2Pixel (*new*)	Finds position for caret given a pointer to text in a buffer and intercharacter spacing

Table 9-3. The Script Manager routines (continued)

Category	*Routine Name*	*Purpose*
	Pixel2Char	Finds nearest character in buffer given a pixel width
	NPixel2Char (*new*)	Finds nearest character in buffer given a pixel width and intercharacter spacing
	DrawJust	Draws text in the current location, font, style, and size
	NDrawJust (*new*)	Draws text in the current location, font, style, size, intercharacter spacing, and scaling factors
	FindWord	Returns word boundaries in buffer
	NFindWord (*new*)	Returns word boundaries in buffer using 'itl2' resources
	HiliteText	Selects characters to be highlighted
	MeasureJust	Measures fully justified text
	NMeasureJust (*new*)	Measures fully justified text given intercharacter spacing and scaling parameters
Formatting text	FindScriptRun	Separates text into blocks using the same script
	GetFormatOrder	Gets the order in which format runs should be drawn based on line direction of the text
	PortionText	Calculates how to allocate white space when justifying text
	NPortionText (*new*)	Calculates how to allocate white space when justifying text given intercharacter spacing and scaling parameters
	StyledLineBreak	Finds where to break a line of text on a word boundary
	VisibleLength	Gets the length of text excluding the trailing white space
Modifying text	LowerText	Converts text to lowercase using international resources (also known as LwrText)
	StripText (*new*)	Strips diacritical characters from text using international resources
	StripUpperText (*new*)	Strips diacritical characters from text using international resources, and converts remaining characters to uppercase
	Transliterate	Converts characters from one script to another
	UpperText	Converts text to uppercase using

Table 9-3. The Script Manager routines (continued)

Category	*Routine Name*	*Purpose*
		international resources (also known as UprText)
Substituting text	ReplaceText (*new*)	Replaces all occurrences of a target string with a replacement string
Truncating text	TruncString (*new*)	Checks that an Str255 string fits into the specified pixel length, and truncates the string if necessary using international resources
	TruncText (*new*)	Checks that text specified by a pointer and count fits into the specified pixel length, and truncates the string if necessary using international resources
Tokenizing text	IntlTokenize	Breaks text into lexical tokens using international resources
Working with dates and times	InitDateCache	Prepares for conversion process
	String2Date	Parses text into dateTime record
	String2Time	Parses text into dateTime record
	LongDate2Secs	Converts time in LongDateRec format to LongDateTime format
	LongSecs2Date	Converts time in LongDateTime format to LongDateRec format
	ToggleDate	Modifies a LongDateRec by incrementing or decrementing fields
	ValidDate	Validates a LongDateRec
Working with locations	ReadLocation	Gets the current geographical information from parameter RAM (longitude, latitude, and time zone)
	WriteLocation	Sets the current geographical information from parameter RAM
Converting numbers	Str2Format	Converts a string into a canonical number format using international resources
	Format2Str	Converts a canonical number format into a string using international resources
	FormatX2Str	Converts a SANE floating-point number to a string using international resources
	FormatStr2X	Converts a string to a SANE floating-point number using international resources

Let's now look at the new Script Manager routines. The System 7 version of the Script Manager provides new versions of several key routines: **NChar2Pixel, NPixel2Char, NDrawJust, NMeasureJust, NFindWord,**

and **NPortionText**. These routines differ from their predecessors, which are still supported for compatibility, because these new routines work with intercharacter spacing, with multiple style runs over multiple lines, and with scaling parameters.

StripText and **StripUpperText** are new routines that your application can call to strip out diacritical characters and convert characters to uppercase, respectively.

Call **ReplaceText** to substitute all occurrences of a target string with a new string. Call **TruncString** or **TruncText**, depending on the way in which the text is stored internally, to decide if a string is too long to fit into the specified number of pixels. These two routines are useful in managing your application's user interface.

Last, the **GetEnvirons**, **SetEnvirons**, **GetScript**, and **SetScript** routines, while not new with System 7, do have some new features. These routines all require a verb, which is simply an integer. This verb specifies the information that you want to set or get. The Script Manager maintains some globals for all scripts that are accessed by the **GetEnvirons** and **SetEnvirons** routines. It also maintains some local variables for each active script system. These are accessed by the **GetScript** and **SetScript** routines. Refer to the Worldwide Software Overview chapter of *Inside Macintosh*, Volume VI for the details.

▶ Compatibility and the Script Manager

Call **Gestalt** with a selector of gestaltScriptMgrVersion to get the version of the Script Manager. Call **Gestalt** with a selector of gestaltScriptCount to get the number of script systems installed on the current machine.

▶ Improvements to International Features

Improvements in international features fall into two categories: improvements to the International Utilities Package and to the international resources. Although they have changed considerably since then, both were part of the first version of the Macintosh operating system.

▶ The International Utilities Package

The International Utilities have always included a set of routines for comparing two strings. With System 7, another set of these string comparison routines has been added, with this new set allowing your application to explicitly specify which 'itl2' resource to use in making the comparisons. The old set of routines, which are fully supported in System 7, were named

IUCompString, **IUMagString**, **IUEqualString**, and **IUMagIDString**. The new set of routines are named **IUCompPString**, **IUMagPString**, **IUEqualPString**, and **IUMagIDPString**. The letter *P* in these names stands for *parallel*, as in parallel routines.

Several new routines make it easier to compare strings in different scripts. These routines use the 'itlm' resource discussed below. Call **IUScriptOrder**, passing two script codes, and it returns with their sorting order. Call **IULangOrder**, passing two language codes, to indicate in which order the two specified languages should be sorted. These two calls establish the sorting order of scripts and languages. Call **IUStringOrder**, passing two strings, along with their script and language codes, and it returns their proper sorting order. Call **IUTextOrder**, passing two strings defined by a pointer and a length, along with their script and language codes, and it returns their proper sorting order.

The **IUGetIntl** and **IUSetIntl** routines allow your application to get and set the 'itl0', 'itl1', 'itl2', and 'itl4' resources. These two routines, which have previously been part of the operating system, are now joined by two new routines, **IUClearCache** and **IUGetItlTable**. Call **IUClearCache** to clear the cache containing 'itl2' and 'itl4' resources. These resources contain information on how to calculate word breaks, compare strings, sort strings, and convert strings to numbers, and they are shared by all active applications. If your application supplies additional 'itl2' or 'itl4' resources, then it should call **IUClearCache** when it is finished using these resources. In effect, this call forces the system to get these resources from the System file the next time they are required. Call **IUGetItlTable**, specifying a script code and a table code (for the word select break table, the word wrap break table, or the number parts table), to get the handle to the resource to which the table belongs and the offset to the specified table in that resource.

The **IUDateString** and **IUTimeString** routines allow your routine to get a string containing the date and time, respectively. These routines, previously part of the operating system, are joined by a new pair of routines named **IULDateString** and **IULTimeString**. These new routines return the same strings as their similarly named predecessors, but take input data in the form of a LongDateFormat instead of the format returned by the **GetDateTime** routine.

▶ The International Resources

The international resources are part of the system software and are kept in the System file. These resources collectively provide the system software and your application with all the information required for customizing the

software for a country, language, and writing system in use at runtime. They are defined in the Rez format in a file provided as part of MPW named *SysTypes.r*. Let's briefly look at each of these resources to see what they contain. The international resources have grown in number and complexity since the Macintosh was first introduced—from two international resource types defined then to 14 different types now. To understand the changes System 7 brings to these resources, let's look at each resource type in turn, reviewing both the old and new contents.

▶ The 'itlc' Resource

Each system has only one 'itlc' resource which always has resource ID 0. This resource contains the following, among other information:

- The system script code
- The preferred region code
- Script Manager flags specifying whether the keyboard icon should always be displayed and whether to use a split caret for a bidirectional script

The preferred region code has been added to this resource in System 7 (Table 9-1 gives a list of region codes). The region code identifies a particular localized version of the Macintosh system software. The Script Manager maintains this value in a global named **smRegionCode**.

▶ The 'itlb' Resource

An 'itlb' resource is attached to each script system and specifies the following information:

- The resource IDs for associated 'itl0', 'itl1', 'itl2', 'itl4', 'KCHR', and 'kcs#' resources
- The default language code
- The number and date representation codes
- Information used by the Script Manager to initialize this script system

Additional information has been added to this resource, but to preserve compatibility with previous versions of this resource, the new informa-

tion has been added to an extension record. The new information includes the following.

- Default 'FOND' ID and font size for the system font, application font, and Help Manager
- Default 'FOND' ID for a monospace font, a variable-width font, and a small-variable font
- A list of styles that this script supports
- A style to indicate aliases

▶ The 'itlm' Resource

The 'itlm' resource, new with System 7, specifies the sorting order for script codes, language codes, and region codes in a set of three tables. The first table contains pairs of script codes and language codes, listed in the preferred order of script codes. The second table contains the same information, but it is listed in the preferred order of language codes. The third table contains pairs of region codes and language codes, listed in the preferred order of region codes.

▶ The 'itl0' Resource

The 'itl0' resource, previously known as the 'INTL' resource ID 0, contains the short formats for dates, times, numbers, and currency. Each script system has at least one 'itl0' resource, the default 'itl0' resource being specified in the script's 'itlb' resource. This resource also contains a region code.

▶ The 'itl1' Resource

The 'itl1' resource, previously known as the 'INTL' resource ID 1, contains the long formats for dates, times, numbers, and currency. It also contains the names of the days, the names of the months, and a region code. Each script system has at least one 'itl1' resource, the default 'itl1' resource being specified in the script's 'itlb' resource.

The format of this resource assumes that all calendar systems have seven days and twelve months. Unfortunately, this isn't true, so the resource has been extended with System 7 to include additional information. To maintain compatibility with the previous format, the new information was added to the end of the resource after first putting a "magic" word in front of the new data. Any system software or development tools

(such as DeRez) that knew about the 'itl1' resource were modified to look for this magic word.

The new 'itl1' resource contains the following additional information:

- A version number and a format number for the extended resource format
- The calendar code (refer to Table 9-2 for a list of all calendar codes) with which this resource should be used
- A list of extra day names and their abbreviations, if more than seven days are used
- A list of extra month names and their abbreviations, if more than twelve months are used
- Additional date separators to be used by the **String2Date** routine

▶ The 'itl2' Resource

Each script system has at least one 'itl2' resource, the default 'itl2' resource being specified in the script's 'itlb' resource. The information contained in this resource includes the following:

- The code and tables for string comparisons
- Tables for case conversion and the removal of diacritical marks by the **Transliterate**, **LowerText**, **UpperText**, **StripText**, and **StripUpperText** routines
- Tables for finding word breaks by the **FindWord** routine
- For non-Roman fonts, a table giving the location of any Roman characters in the font for use by the **FindScriptRun** call

This resource also has a new header, making it easier to locate the various types of information in this resource.

If your application needs different string-comparison behavior, you can create an 'itl2' resource to be used by your application. You might need to do this to support a language that is not supported by the existing software. If you do use your own 'itl2' resource, be sure to call the **IUClearCache** routine (described above), or your resource will linger on.

▶ The 'itl4' Resource

Each script system has at least one 'itl4' resource, the default 'itl4' resource being specified in the script's 'itlb' resource. The information contained in this resource includes the following:

- A character used to mark truncated text (in English it is the ellipsis: ...)
- Code and tables for the **IntlTokenize** routine
- Formatting information for the **Str2Format**, **Format2Str**, **FormatXStr**, and **FormatStr2X** routines
- A new header with offsets to the code and tables

The 'itl4' resource existed before System 7, but it is now documented for the first time.

▶ The 'KCHR' Resource

The 'KCHR' resource contains tables for mapping key codes to character codes. Each script system has at least one 'KCHR' resource, and the default 'KCHR' resource is specified by ID in the script's 'itlb' resource. The 'KCHR' resource for the Roman script has numerous small changes to the mapping tables.

▶ The 'kcs#', 'kcs4', and 'kcs8' Resources

These resources, new with System 7, are color icons for the keyboard. They match the resource ID of their corresponding 'KCHR' resource. The 'kcs#', 'kcs4', and 'kcs8' resources are identical in format to the 'ics#', 'ics4', and 'ics8' resources. In other words, they are icons for 1-bit, 4-bit, and 8-bit displays. The default 'kcs#', 'kcs4', and 'kcs8' resources are specified by ID in the script's 'itlb' resource. Note that keyboards only have small icons (16 by 16) but not large icons (32 by 32). These keyboard icons are used in the Keyboard control panel and in the Keyboard menu.

▶ The 'KCAP' Resource

The 'KCAP' resource contains the physical layout of keyboards, which used to be contained in the Key Layout file installed in the System Folder. Starting with System 7, the 'KCAP' resource is in the System file. Each keyboard has one 'KCAP' resource, whose resource ID matches the keyboard code. The keyboard codes are listed in Table 9-4 (after *Inside Macintosh*, Volume VI).

Table 9-4. Keyboard codes

Keyboard Type (hex)	*Keyboard Name*
$1	Standard ADB
$2	Extended ADB
$3	Macintosh 128K and 512K (U.S.)
$103	Macintosh 128K and 512K (International)
$4	ISO Standard ADB
$5	ISO Extended ADB
$6	Portable Standard
$7	Portable ISO
$B	Macintosh Plus

▶ The 'KSWP' Resource

The 'KSWP' resource contains a table of entries specifying which key combinations can be used to switch from one script to another.

▶ The 'itlk' Resource

The 'itlk' resource contains key-code mapping information that is used to make the international keyboard layouts work on all keyboards. This resource is used by the **KeyTrans** routine. The 'itlk' resource existed before System 7, but is documented for the first time.

▶ Writing International Software

Software *localization* is the process of adapting software to a particular country or language. Converting an application to another European language is relatively easy to support, although there are some nontrivial details. If you follow Apple's guidelines (see "Get Info" for references), your application can be easily localized into most European languages by translators who have little technical knowledge, because most of the European languages behave similarly to English.

Warning ▶ Some kinds of software, even if localized, may not be useful in other countries. For example, accounting methods are quite different in Europe, so most American accounting packages would require considerably more work to be useful in Europe.

To fully support localization, you need to use the Script Manager.

Working with KanjiTalk or the Arabic script system is more difficult than simply ensuring that your application will work in German or Portuguese, because Japanese and Arabic use very different writing systems. If your application is not text-intensive—that is, it isn't a word processor or publishing system—you won't have to do a lot of work to use the Script Manager. Programmers developing word processors and publishing systems, on the other hand, will have to do quite a bit of work to ensure that their applications can be used with a variety of script systems.

Be aware also that localized applications do not necessarily support the use of multiple languages and scripts. To allow users to write in multiple languages and scripts requires more work than just supporting the Script Manager.

▶ Conclusion

In this chapter, you've looked at the latest version of TextEdit, which provides a standard set of text-editing capabilities. You've also looked at the Script Manager and the International Utilities Package, which provide services that can ensure that your application can be used in countries outside the U.S.

The computer market in the U.S. is growing, but at a slower rate than in past years. The computer markets in most of the rest of the world are growing quickly. If your application can be localized and can support non-Roman script systems, then your application can participate in the fast growth of the Macintosh in these other countries.

Get Info ▶

For more information on TextEdit, refer to the TextEdit chapter of *Inside Macintosh*, Volume VI. You may also want to refer to the TextEdit chapters of *Inside Macintosh*, Volumes I, IV, and V.

For more information on internationalization and the Script Manager, refer to the Worldwide Software Overview chapter of *Inside Macintosh*, Volume VI. Another useful reference is *Macintosh Worldwide Development: Guide to System Software*, available from APDA, which explains the basics of script systems, the Script Manager, the International Utilities Package, and the international resources.

The Software Licensing group at Apple licenses all the localized versions of the Macintosh operating system at reasonable rates. If your application uses the Script Manager, then you should get a variety of localized systems to test with. Apple has also made all the international systems and keyboards available to companies that are members of the Apple Partners and Associates programs.

10 ▶ The Data Access Manager and the Data Access Language

▶ Introduction

The database management systems that run on mainframes and minicomputers are large, complex applications, which provide a way to store enormous amounts of complex data and ways for many users to access this data simultaneously. This cannot be done—yet—on microcomputers. Many organizations have large amounts of data stored on larger host computers. The Data Access Language and the Data Access Manager provide a simple, relatively easy way for Macintosh applications to tap into these vast amounts of data.

In this chapter, you'll first look at why the Data Access Language (DAL) was invented and the types of tasks for which it can be used. You'll look at this language and its relation to the SQL database access language.

Following this, you'll look at the architecture of the Data Access Manager (DAM). This component of the operating system provides an API to access databases and other data. Although it can be used with the Data Access Language, this manager is a general interface and can be used with other access languages as well. In the future, it will be used to access data located on various platforms and in various databases besides those supported today. You will look at the high-level and low-level routines provided by the Data Access Manager. You'll also look at the question of when to use the DAL and when to use proprietary database access methods.

▶ Why the Data Access Language?

Note ▶ A database management system (DBMS) is a collection of software (and sometimes hardware) that provides a set of services for storing and retrieving data. A database is a file of data maintained by a DBMS.

Imagine you're writing a system that needs access to two databases. One of them is an Ingres database running on VAX under VMS, and the other is an IBM DB2 database running on a mainframe under MVS. To write the software for this system, you'd need to understand and write code to do the following:

1. Set up a communications link from a Macintosh to both the VAX and the IBM mainframe. You might have to communicate with the VAX using AppleTalk for VMS, and use a 3270 data stream protocol to talk with the mainframe. The code for these links must be able to initialize the communications link and handle any errors that might arise.
2. Start up the appropriate DBMS on the host. In this chapter, the host will also be called the data server.
3. Open the user's database.
4. Retrieve the information the user wants to see, performing any intermediate calculations that might be needed.
5. Close the database.
6. Quit the database management system.
7. Shut down the communications channel.

Whew! If it sounds like you'd have to write a lot of code, you're right: "Each step" would require a lot of code. Fortunately, the Data Access Manager and the Data Access Language can simplify this whole process as follows:

1. Call **InitDBPack** to initialize the Data Access Manager.
2. Call **DBNewQuery** to create a query record.
3. Call **DBStartQuery** to complete the query record, initiate a session with the host, start up the DBMS, open the database, send the query, and start executing the query.

4. Call **DBGetQueryResults** to retrieve the results from the data server.
5. Call **DBResultsToText** to convert the data returned from the database into text strings.
6. Call **DBDisposeQuery** to deallocate the data structures used in making the query.
7. Call **DBEnd** to shut everything down.

This is a lot simpler than the previous example, but it depends on several other large pieces of software: the software to access the network, the DAL server, and the database adapter to the vendor's DBMS. By building on top of a layered architecture, your job as an application developer is a lot easier, so long as all the layers you need are there.

Figure 10-1 illustrates what you can do with the Data Access Manager. This example has three Macintosh applications: a spreadsheet, a report writer, and a word processor. Each of these applications can use any available communications link available on their Macintosh to connect with a host computer. Minicomputers, such as a DEC VAX, or mainframes, such as an IBM mainframe, are supported by the Data Access Manager. Several different DBMSs, each from a different vendor, are supported on these machines. By use of the DAM, a user has access to information on a variety of databases using various communications channels, various hosts, and various DBMSs. This all happens through a single interface provided by the Data Access Manager.

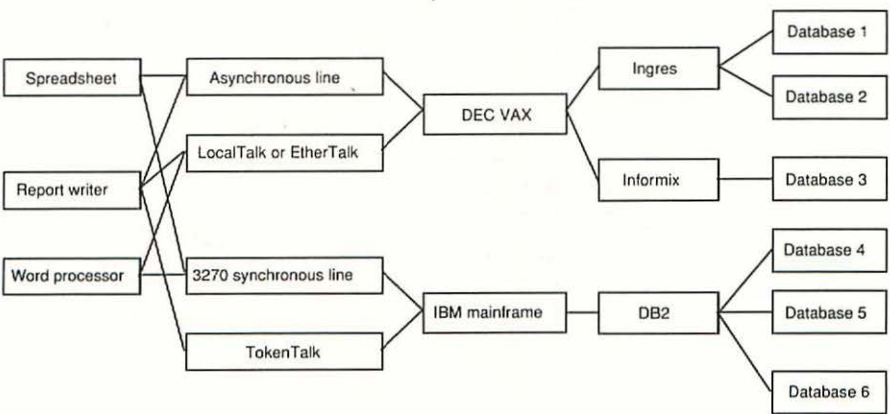


Figure 10-1. Various uses of the Data Access Manager

You can use DAL and the DAM with your application to ensure that users are always using the latest data. Because these tools can hide the details of navigating through a network to a host and navigating through a database to a set of data, nontechnical users can use large, complex databases knowing little of the details.

▶ The Architecture of the Data Access Manager

The Data Access Manager provides a generic application programming interface to databases and other data sets using the client/server model. In System 7, the Data Access Manager (DAM) will be used primarily in conjunction with the Data Access Language (DAL). It is important to realize, however, that these two are not the same. To understand this, you must look at the architecture of the Data Access Manager and at how it is used.

Figure 10-2 illustrates the architecture of the Data Access Manager. In the Macintosh, three software components work together. An application calls the routines of the Data Access Manager to send a query to a host or data server. The query might be written in Data Access Language, but it could be expressed in any language.

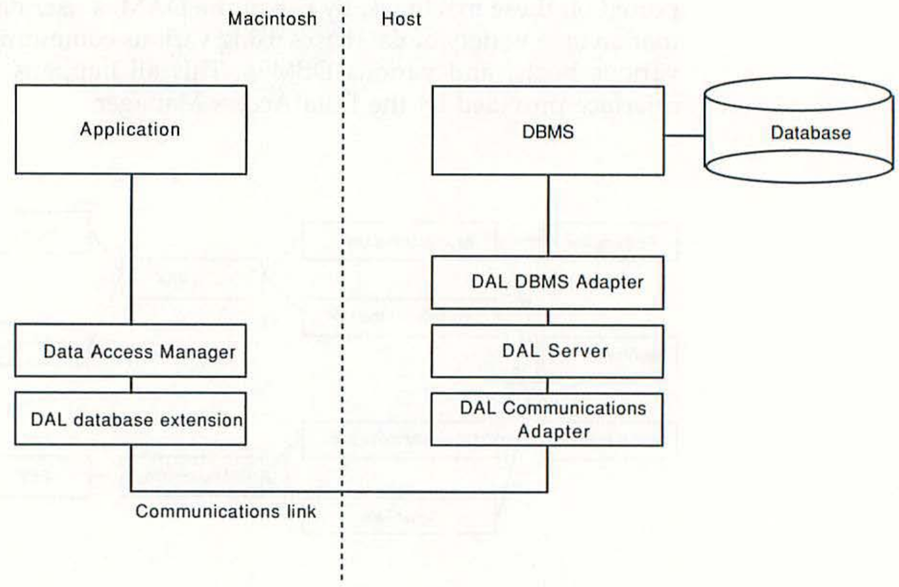


Figure 10-2. The architecture of the Data Access Manager

The Data Access Manager is told to use a particular *database extension* when making this query. A database extension is responsible for setting up a communications channel with a host (if one is required). System 7 ships with one database extension for use with the Data Access Language. Then, by communicating over the channel with a peer process on the host, the database extension sends the user's name and password and then the query. An extension for a local database may not require all this work, but the DAL extension does have to do it.

Note ▶ Currently, all data servers are on minicomputers or mainframes. Future database extensions will support local databases on the user's Macintosh, as well as databases located on AppleTalk file servers.

The DAL database extension works with a peer process on the data server. The peer process consists of four major components:

- An operating-system adapter
- A network adapter for the particular communications protocol in use
- A DAL server that handles the generic aspects of query processing
- A database adapter that translates the query from DAL into the proprietary query language needed by the DBMS managing the database to be queried

The server process on the host communicates with a DBMS process and passes the user's query to it. The DBMS returns the results of the query to the server process, which translates the results from the proprietary formats of the DBMS into a generic format supported by DAL. Next, the server process sends these results back to the database extension, which, in turn, passes the results back to the DAM when the application requests the results. DAL generally can start returning results before the query has been completely processed by the DBMS on the host.

Only a small amount of code is needed in the Macintosh to provide these services. The only Macintosh code involved besides your application is the Data Access Manager, which is part of the operating system, and any database extensions, such as the DAL extension provided as part of System 7. Some communications code is also needed on the Macintosh. On the other hand, the host machine requires a great deal of code. In addition to one or more database management systems, the DAL server code is a substantial application by itself.

An important aspect of this process is that the Data Access Manager and the database extension do not look at the user's name, password, or query. The server process in the host does, and so "errors in query" can be found only by that process or the DBMS. The software in the Macintosh does not do any error checking on the query.

▶ Queries and the Data Access Manager

In this chapter, the term *query* means simply a set of one or more commands to a database. These commands might retrieve data from a database, or they might store data in a database. In other words, do not assume that a query always retrieves data; it might be performing some other task.

Queries come in two forms: *query documents* stored on disk, and queries built in memory in real time. In either case, a query is always tied to a particular database extension because different extensions may require additional or different data.

Query documents, because they only contain a set of commands for a database, are application-independent. Any application that can use the DAL and DAM can make use of a query document. A user could use the same query document when running a spreadsheet or a word processor. The results returned will be the same in either case, no matter which application was used (unless the data in the database have changed in the meantime). Query documents provide a simple but powerful way for any application to retrieve data.

Queries built in real time require more work than query documents, but they are much more flexible than query documents. Data-intensive applications will build queries at runtime, whereas most applications will use query documents.

▶ What Do You Need to Run DAL?

If you are developing, or running, an application that uses DAL, you'll need several different pieces of software. First, you'll need a host machine with at least one DBMS installed. This machine must also have some communications hardware and software that can communicate with Macintosh computers. Finally, you'll need the DAL server software for this machine and DBMS. Hardware, DBMS, and communications requirements for running the DAL server software are listed in Table 10-1.

On the Macintosh, in addition to your application, you'll need the appropriate communications software and hardware to communicate with the host machine.

Note ▶ Note that you must be running System 7 on this Macintosh. You can write applications that use DAL under System 6, but in this case you must license the DAL software from Apple to link into your application.

Table 10-1. Requirements for running DAL on host machines

Host Type	*Supported Communications*	*Supported DBMS*
DEC VAX (running VAX/VMS)	AppleTalk for VMS (Apple), LanWORKS (DEC), asynchronous communications	Ingres, Informix, Oracle, Sybase, DEC rdb
IBM mainframe (running VM/CMS)	3270 data stream protocol	IBM SQL/DS
IBM mainframe (running MVS/TSO)	3270 data stream protocol	DB2

▶ The Data Access Language

The Data Access Language (DAL) was originally known as CL/1. This language and the precursor to the Data Access Manager were developed by Network Innovations—founded in 1984, acquired by Apple in 1988, and now a wholly-owned subsidiary of Apple.

DAL provides a generic language for relational databases, although you can use most of the language to interact with other types of databases. The C language was used as the model for DAL. DAL includes features of SQL (Structured Query Language), which is a standard query language for interacting with relational database management systems. SQL originated in work done by IBM on databases in the early 1970s. Since that time, the language has been widely adopted by virtually all vendors of relational DBMSs and has also become an ANSI standard.

SQL is primarily a data manipulation language. As such, it provides facilities for interacting with a database, but does not provide capabilities for handling communications, program control, arithmetic and logic, and data output—SQL was never intended for those purposes. SQL statements are embedded in applications written with COBOL, C, Fortran, and other programming languages that provide those capabilities and more. SQL does not provide a standard way of handling errors or a standard set of error codes. DAL provides both.

DAL is to database management systems as PostScript is to printers. PostScript provides a platform-independent way for any application to print documents of all kinds. Similarly, Data Access Language provides a platform-independent way for any application to interact with a database.

The language statements of DAL can be divided into three groups: data manipulation (this is the portion of the language that resembles SQL), program control, and output. Table 10-2 lists all the DAL statements.

Table 10-2. The Data Access Language statements

Category	*Statement*
Data manipulation	OPEN DBMS
	CLOSE DBMS
	USE DBMS
	OPEN DATABASE
	CLOSE DATABASE
	USE DATABASE
	OPEN TABLE
	CLOSE TABLE
	USE LOCATION
	SELECT
	FETCH
	DESELECT
	UPDATE
	DELETE
	INSERT
	LINK
	UNLINK
	COMMIT
	ROLLBACK
	DESCRIBE DBMS
	DESCRIBE OPEN DBMS
	DESCRIBE DATABASES
	DESCRIBE OPEN DATABASES
	DESCRIBE TABLES
	DESCRIBE LINKSETS
	DESCRIBE COLUMNS
	EXECUTE
Program control	DECLARE
	UNDECLARE
	SET
	IF
	SWITCH
	GOTO/LABEL
	WHILE
	DO
	FOR
	FOR EACH

Table 10-2. The DAL statements (continued)

Category	*Statement*
	BREAK CONTINUE PROCEDURE CALL RETURN
Output	PRINT PRINTROW PRINTALL PRINTF PRINTINFO PRINTCTL ERRORCTL

DAL also provides the following elements common to most programming languages:

- Compound statements—A sequence of DAL statements are grouped together and executed sequentially when bracketed with { and } characters.
- Comments—Anything between the /* characters and the */ characters is ignored by the DAL interpreter. Comments can be nested.
- Procedures—The PROCEDURE and CALL statements provide this capability in the language.
- Functions—DAL provides a considerable number of built-in functions, such as $len, $substr, $colname, $cols, and $rows.
- Identifiers—Identifiers begin with a letter, are 31 characters or less in length, and are case-insensitive. Identifiers can contain letters, numbers, or underscores, but no space characters.
- Variables—Each variable has a unique identifier. The three types of variables are system variables (maintained by the DAL interpreter), external variables (declared outside a DAL procedure definition), and local variables (declared inside a DAL procedure definition).
- Data types—DAL provides a wide variety of data types. The data types shared with the Data Access Manager are listed in Table 10-3. DAL also provides three additional types useful in writing DAL programs: CURSOR (identifying an active set of rows), OBJNAME (for storing DAL identifiers), and GENERIC (for storing any type of data).

- Literals—Literals for most of the data types listed in Table 10-3 can be written in the obvious manner.
- Expressions—Expressions use the operators listed in Table 10-4 to combine literals, variables, and other identifiers for calculations. You can use parentheses to alter operator precedence rules. The DAL interpreter also converts the data type of an expression to match the type of result, and it provides a full set of type conversion operators.

Table 10-3. Supported data types in Data Access Language

Data Type	*Length (bytes)*	*Description*
typeBoolean	1	TRUE (1) or FALSE (0)
typeSMInt	2	Signed short integer
typeInteger	4	Signed long integer
typeSMFloat	4	Signed small floating-point number
typeFloat	8	Signed floating-point number
typeDate	4	2 bytes for year, 1 byte for month, 1 byte for day
typeTime	4	1 byte for hour, 1 byte for minutes, 1 byte for seconds, 1 byte for hundredths of a second
typeTimeStamp	8	typeDate and typeTime
typeChar	any	Fixed-length string (defined by the data source)
typeDecimal	any	Packed decimal string
typeMoney	any	Same as typeDecimal, but always has 2 decimal places
typeVarChar	any	A variable-length string of characters (with a length)
typeLongChar	any	An unbounded sequence of characters
typeVarBin	any	A variable-length string of bytes (with a length)
typeLongBin	any	An unbounded sequence of bytes of binary data

▶ Aspects of Using DAL

DAL is based on a model of databases having three levels: the database management system (DBMS), the databases accessible from the DBMS, and the tables in a database. This enables DAL to support nonrelational DBMSs, with a lot of work going into the server software.

Table 10-4. Data Access Language operators

Operator Type	*Operators*
Arithmetic	+ (addition)
	− (subtraction and negation)
	/ (division)
	* (multiplication)
String concatenation	+
Comparison	= or == (equality)
	!= or <> (inequality)
	< (less than)
	<= (less than or equal to)
	> (greater than)
	>= (greater than or equal to)
Logical	AND
	OR
	NOT

Some of the DAL statements are of special note. Use the DESCRIBE verbs to find out which DBMS to use, what security requirements it has, and so on. If appropriate, you can present your user with a list of available DBMSs, then with a list of databases accessible from that DBMS, and finally with a list of tables in that database. Note that the language has no provision for listing all the hosts—you'll have to write the code if you need that capability.

The EXECUTE command passes the remainder of the statement directly to the DBMS and is not touched by the DAL interpreter. You can use this statement to issue commands that are not supported by DAL, such as unique or proprietary commands.

Use the COMMIT and ROLLBACK statements to manage *transactions*—that is, a set of one or more commands that should be performed as a unit. By using these commands for more than just updates, you can improve the performance of your application and the host DBMS. The rule to follow is to use the COMMIT command early and often. Also, close cursors as soon as you are done with them.

The OPEN TABLE and CLOSE TABLE statements do not affect a relational DBMS and are ignored by the DAL interpreter in such a case. They do, however, have a significant effect on the performance of nonrelational DBMSs. You should therefore use these statements whenever a query can be made against a nonrelational DBMS.

By setting the $fetchmode system variable, your application can control the performance of the SELECT, FETCH, and FOR EACH statements.

You can select from four levels. The levels are as follows, starting at the mode with the highest level of performance:

- Read-only—Your application can read each item once in the forward direction only. That is, it cannot return to a previous item without executing the query again. In this mode, you cannot get the total number of rows before they are all available.
- Update—Your application can read or update each item once in the forward direction only. That is, it cannot return to a previous item without executing the query again. In this mode, you cannot get the total number of rows before they are all available.
- Scrolling—Your application can read or write each item and can proceed in either the forward or backward direction. In this mode, you cannot get the total number of rows before they are all available.
- Extract—Your application can read or write each item and can proceed in either the forward or backward direction. In this mode, you can get the total number of rows before they are all available.

Error codes are translated by the DAL interpreter into the standard DAL set of error codes, which are compatible with the error codes used by IBM's DB2 relational DBMS. Because errors for all supported DBMSs are converted into a standard set, your application does not have to deal with or know anything about this aspect of DBMSs.

DAL cannot hide certain details of each DBMS from your application. Several areas your application may have to deal with are null and missing data, the collating sequence of characters, support for international character sets, aggregate operators, and GROUP BY rules. These areas are all characteristic of the low-level features of each DBMS and cannot be translated without a considerable amount of work.

▶ Security Considerations

Both the Data Access Language and the Data Access Manager use the standard security measures of the host operating system or the database management systems running on the host. Several levels of security are involved in talking to a particular database from a particular DBMS on a particular host computer.

Access to the host happens over a network, provided that the user has access to the network. The user must provide the host with a valid user name and password. If these are not valid, the connection fails and the Data Access Manager returns with an error.

The user must also provide the DBMS with a user name and password. Again, if these are not accepted by the DBMS, the connection fails and the Data Access Manager returns with an error.

Finally, the user may also have to provide the particular database with a user name and password. As before, if these are not valid, the connection fails and the Data Access Manager returns with an error.

The current versions of the Data Access Manager and Data Access Language have one potential security problem: The user name and password provided by the user are not encrypted. You can monitor the communications path between the user and the host computer and see the user name and password.

▶ Examples of Data Access Language

Here are several fragments of DAL to give you some flavor of how you can use the language. The first example opens a marketing database using a name and password, selects all the people who have an interest level greater than 4 and returns their addresses, and deletes all uninterested contacts and closes the database.

```
open ingres database "mkting" as user  "snidley" password "bah";
select name, address, city, state, country, zipcode from
     contacts where interest_level > 4 group by zipcode;
printall;
delete from contacts where interest_level = 1;
close database "mkting";
```

The next example opens the orders database and inserts a new record into the order_main table. Then it inserts three rows into the order_detail table and closes the database.

```
open db2 database "orders" as user "nell" password "good";
insert into order_main (name, account_no, order_no) values
     ("Acme Company", "A1787", "11-98343");
insert into order_detail (order_no, prod_code, prod_qty) values
     ("11-98343", "M5011", 13);
insert into order_detail (order_no, prod_code, prod_qty) values
     ("11-98343", "M5650", 37);
insert into order_detail (order_no, prod_code, prod_qty) values
     ("11-98343", "M0400", 37);
close database "orders";
```

The next example asks for information about all the databases and their tables. This example assumes that the user has already connected with a host.

```
/* db will hold the names of all databases */
declare cursor db;
declare boolean done = $false;

/* Get all the names of databases */
describe ingres databases into dbs;
printall;

/* For each database */
while (not done)
     {
     fetch db;
     if(sqlcode <0)
          {
          done = $true;
          continue;
          }
     /* get info on all tables */
     describe tables of db;
     printall;
     }
```

When using DAL with the Data Access Manager, you will often build up queries in real time. In other cases, the user will choose a prebuilt query stored in a query document. Refer to the section on using the DAM routines for more information on how to do this.

▶ When to Use DAL and When to Use a DBMS Directly

The question of when to use DAL and when to use the DBMS directly does not have a simple answer. Some rough guidelines follow.

Use DAL when the data you'll be accessing fall into one of the following categories:

- Live on more than one database managed by a variety of vendor's DBMSs—Having to access more than one DBMS often qualifies here.
- Live on a wide variety of hosts—Having more than one vendor's host usually qualifies here; for example, a DEC VAX and a Data General MV. One exception to this guideline: If the data you're after is in the same vendor's DBMS running on different hosts, you may have an easier time staying within that vendor's system.

- May be known only at runtime—Although some DBMS vendors provide excellent tools for accessing their databases on an ad hoc basis, the third-party tools for doing ad hoc queries with DAL are hard to beat.
- Easily specified or accessed by the variety of third-party tools that understand DAL—One clear advantage of DAL over proprietary database access tools is that numerous third-party tools can talk to your databases. You can often use these tools straight out of the box by specifying a query (and setting up the network access, installing the DAL server software on the host, and so on).

Use a DBMS directly when the data you'll be accessing fall into one of the following categories:

- Live in one or more databases managed by a single vendor's DBMS—DBMS vendors usually provide tools to access databases managed by their DBMS on a variety of different types of host computers. For example, if all the databases you will be accessing are Oracle databases, then the tools provided by Ingres may be exactly what you need.
- Live on a single vendor's host computer—Having more than one vendor's host usually qualifies here. One obvious exception here: If the data you're after is in the same vendor's DBMS running on different hosts, you may have an easier time staying within that vendor's system.
- Are known at compile-time—The highest performance from database access tools is usually available using tools provided by that vendor. Transaction-based application systems should probably never go through DAL unless high performance is not a requirement.

Remember that these are general guidelines. You can get good performance from DAL if you tune both your application and queries. You should also look at tuning the host system and the DBMS for optimal performance.

The question of performance is never a simple one. You may need to do some initial prototyping and benchmarking around your application with DAL and proprietary tools to give you enough information about performance so you can make an informed selection.

▶ The Future of the Data Access Language

DAL is now part of the Macintosh operating system. More and more applications will support DAL. More and more DBMS vendors can be expected to support DAL. What will the evolution of DAL and the Data Access Manager bring? Some obvious changes to look for are as follows:

- Improved performance
- Support for more host computers
- Support for more host database management systems, perhaps including hierarchical and network model DBMSs and flat-file management systems (such as DEC's RMS) in addition to other relational DBMSs
- Support for more communications protocols

These changes will not affect most applications that use DAL or the DAM; however, they will increase the range of these two tools to other platforms, so users of your application will potentially have more reasons for accessing databases through your application.

Other changes that you can expect to see in the future include the following:

- A porting kit for the DAL server, allowing anyone to port the server to another host or to support another DBMS
- A software toolkit and documentation for developing database extensions on the Macintosh, allowing local Macintosh databases to be supported
- Support for additional data types, such as blobs (large unstructured chunks of data which could be used for storing documents and pictures in databases)
- More sophisticated support for stored procedures

Some of these changes will require alterations in applications that use DAL and the DAM; other changes won't require any alterations.

▶ Using the Data Access Manager

The Data Access Manager (DAM) provides two sets of routines: a high-level set of a calls and a low-level set of calls. You can use the calls from one set or you can use calls from both sets, since the two sets are compati-

ble. You'll look at how to use these two sets of calls after first looking at query documents.

The high-level calls are designed for applications that are not data-intensive, such as word processors. These calls are quite easy to use, but offer little control or flexibility. High-level calls use query documents created using another application. Apple recommends that the high-level calls be used for retrieving data, but not for uploading data, because the high-level calls cannot verify that the uploaded data was received by the host.

The low-level calls give you a great deal of control over the process of working with a DBMS, but the price you pay is that you will have to write more code than if you chose the high-level set of calls. You will also need to understand more about databases to use these routines most effectively and to get the best performance from the communications network, the host, and the DBMS.

You can use the low-level calls and most of the high-level calls asynchronously. To make an asynchronous call, pass a pointer to an asynchronous parameter block record. You'll need to fill out some values in the record before passing it. Check the result field in the record to find out when the call has completed.

▶ Compatibility and the Data Access Manager

Call **Gestalt** with a selector of gestaltDBAccessMgrAttr to inquire about the Data Access Manager. The only attribute returned from this call under System 7 tells you whether the Data Access Manager is present.

▶ Query Documents

A query document is a document that contains a set of commands for use with the Data Access Manager. Each document is written for use with one or more database extensions. System 7 comes with one database extension, written to handle queries written with the Data Access Language. Note that the term *query* in this context does not mean that it can only be used to retrieve data. Here, the term *query* refers to a set of one or more commands (and any associated data) that will be executed by a DAM server. Generally, a query document can be used by many, if not all, applications that support query documents.

Query documents are files of type 'qery'. If your document can create or modify such documents, then you may want to create an icon for them. Each query document contains a set of resources:

- A 'qrsc' resource—Contains the resource IDs for all other resources in this file.
- An 'STR#' resource—Contains the name of the database extension to be used with this document, and (optionally) the name of the host, the user name, password, and connection information. Some of the parameters may be null strings; in such a case, these strings will be constructed at runtime.
- One or more 'wstr' resources—Contain strings of commands and data to be sent to the data server.
- An optional 'qdef' resource and other associated resources—A 'qdef' resource contains code that will present a user with a dialog box prompting him or her for information used in making the query.

The purpose of a 'qdef' resource is to build a query record in memory. A query record contains all the information related to a query. The DAM will construct a query record using information contained in the 'qrsc' and 'STR#' resources. If a 'qdef' function exists, the DAM will call that function and pass the handle to the query record. That function can modify the query record.

When writing a 'qdef' resource, avoid writing it in an application-specific manner. That is, if possible your query should be usable from any application that can use query documents. Your 'qdef' function will be passed a session ID if a session is already open; otherwise, your function can optionally open a session. You will also be passed a handle to a query record, and you can modify the contents of this record. The 'qdef' must return an error code of noErr, or the query will not be sent to the data server.

A query document can have more than one query. The user or the 'qdef' function will choose a query from the set and add it to the query record. The high-level calls to the Data Access Manager open the query document selected by the user. The 'qdef' function is called by the Data Access Manager to present the user with a dialog box. After the user has selected a query or entered data, the query is sent to the selected data server.

▶ Using the High-Level Calls to the Data Access Manager

To initialize the Data Access Manager, call **InitDBPack**. You must make this call before using any other routine in this manager. It is safe to call this routine more than once; if the DAM is already running, no harm will result.

To create a query record, call **DBGetNewQuery**, passing the resource ID of the 'qrsc' resource to be used. This routine uses the contents of the 'qrsc' resource to construct the query record. If a 'qdef' resource is present, then

this routine will call that code. The resource file containing that resource must remain open until the call to **DBStartQuery** has completed, because other resources from that file will be used in creating the record.

To dispose of a query record once you are finished with it, call **DBDisposeQuery**. This routine frees all memory associated with the record as well.

Call **DBStartQuery** to complete the query record, initiate a session with the host if required, send the query, and ask the data server to execute the query. This routine performs its tasks by using several of the low-level routines. The parameters to pass this routine include a handle to a query record, a pointer to an optional asynchronous parameter block record, and a session ID. If the session ID is 0, then this routine will initiate a session with the host; otherwise, it will use the existing session. You can also provide the address of a status procedure. This routine will call your status procedure as things progress, allowing you to keep the user informed about the progress of the query.

To retrieve the results of your query, call **DBGetQueryResults**. You'll pass several parameters, mostly the same as for **DBStartQuery**. The one additional parameter is the handle to a results record. When this call is completed, the results record will contain as much data as will fit into memory. If there is more data than will fit in memory, then you'll have to make additional calls to **DBGetQueryResults**. You can also specify an optional timeout value for this call. This routine can also be used to retrieve the results of queries submitted with the low-level interface.

The information in the results record is in a binary format. To convert this information to text strings, call **DBResultsToText**, which calls the appropriate result handlers to do this. This routine, and the result handlers, are discussed below.

▶ Using the Low-Level Calls to the Data Access Manager

To initialize the Data Access Manager, call **InitDBPack**. You must make this call before using any other routine in the DAM. It is safe to call this routine more than once; if the DAM is already running, no harm will result from this.

To start a session with a data server, call **DBInit**. This routine returns a session ID, which is required to used most of the other low-level calls. You'll need to pass this routine the name of the database extension to use ('DAL' for the extension provided as part of System 7), host name, user name, password, and an optional string to be passed to the data server.

To conclude a session, call **DBEnd**. This routine closes the communications link between the local machine and the host.

To send a query (or a portion) to the data server, call **DBSend**. You can use this call to send a query in more than one piece, because the data server will append all the strings until you call **DBExec**. The parameters for this call are a session ID and a pointer to the text to be sent to the data server. Since the DAM and the database extension pass this information without looking at it, the data server receives the text unaltered.

To send data to the host as part of a query, call **DBSendItem**. The parameters for this call are a session ID, the data type, the length of the data, the data format, and a pointer to the data.

After sending a query and data to a data server, execute the query by calling **DBExec**, passing it the session ID. Call **DBState** to check on the status of the query, again passing the session ID. If you get an error of rcDBError, then the data server has rejected the query. Call **DBGetErr** to get the error code(s) and error message(s) from the data server. The meaning of these codes and messages is specific to the data server.

Call **DBGetItem** to retrieve data generated by the query from the data server following your call to **DBExec**. You can also use this call to retrieve the same data more than once or to retrieve the characteristics of the next item without having to retrieve the data. You should pass the session ID, a timeout value, a pointer to a data buffer, and the length of the buffer. You also specify the type of the data you expect. If the next item isn't of that type, you'll get an error of rcDBBadType. To accept the next item regardless of the type, specify typeAnyType. You can skip an item by specifying a data type of typeDiscard. To get the characteristics of the next data item, pass a NIL buffer pointer.

DBGetItem returns the data type of the item, the characteristics of the data, and the length of the data in addition to the data itself. This routine sets flags to tell you whether the data item is NULL or if it is the last in a row. The data returned are in a binary format; to convert to text, use the **DBResultsToText** routine (described in the following section).

Call **DBUnGetItem** to reverse the effect in the data server of the last call to **DBGetItem**. This call does not change the local copy of the data. Also, you can use this call only to reverse the last call to **DBGetItem**; you cannot use it to reverse the effect of more than one call.

▶ Using Result Handlers

The information returned in the results record is in a binary format. Call **DBResultsToText** to convert this information to text strings, passing a results record and a pointer to a buffer for the text. This routine calls the appropriate result handlers to accomplish this. There are two sets of result handlers: one set for the current application and a default system set of handlers. If a particular data type has no application result handler, then

this routine calls the system result handler for that type. You can write your own handlers for one or more data types.

System 7 provides a set of result handlers for all data types listed in Table 10-3, with the exception of those reserved for future use. Apple also provides three additional handlers. One handler is called when no other result handler can be called to process the results. The other two handlers are called by **DBResultsToText** after each item. The handler for typeColBreak is called after each item that does not end a row of data; the default handler inserts a tab character. The handler for typeRowBreak is called after each item that does end a row of data; the default handler for this case inserts a return character. These last two handlers are not used anywhere else.

Call **DBInstallResultHandler** to install a new result handler. The parameters for this call are the data type, a pointer to the result handler procedure, and a Boolean that controls whether the result handler should be used for this application only or for all applications. You would use this call to install your own handler.

Call **DBRemoveResultHandler** to remove the current application result handler for the specified data type. You cannot use this routine to remove a system result handler.

Call **DBGetResultHandler** to get the address of either the current system or application result handler for the specified data type. This routine is useful for several purposes. You can use it to get the address of a particular result handler for use with **DBResultsToText**. You should use this routine in conjunction with the **DBInstallResultHandler** routine before installing a new system result handler. When your application is finished, reinstall the previous system result handler using the address returned by the call to **DBGetResultHandler**.

If you are using the low-level call **DBGetItem** to retrieve results, you will have to call the **DBGetResultHandler** result handler directly. First, retrieve the address of the handler using the call. Then call the handler, passing the data retrieved with the **DBGetItem** call.

▶ Using the Low-Level Utility Calls

To halt the execution of a query and reinitialize the data server, call **DBBreak**. You can also use this call to terminate the session. Pass the session ID and a Boolean specifying whether the session should be terminated or the data server reinitialized. Call **DBKill** to kill any asynchronous call. The only parameter to this call is the pointer to an asynchronous parameter block record.

To get information about the specified session, call **DBGetConnInfo**. The information returned by this routine includes the following:

- The name of the host
- The user name (but not the password)
- The connection string
- The name of the network used to connect to the host
- The time the session was started
- The database extension and its session number for this session
- The status of the session

Your application can use this call to find out about concurrent sessions on your local machine by specifying each database extension and a session number. Session numbers are unique for each extension, and the first session number is always 1. By incrementing the session number and checking for valid data, you can find out the status of all other DAM sessions.

Call **DBGetSessionNum** to find out the session number of the session with the specified session ID. You could use this call to get your own session number. Each database extension creates its own session numbers. If more than one extension is active, more than one session can have the same session number.

▶ Human Interface Guidelines and the Data Access Manager

Your application should follow the guidelines recommended by Apple for using the Data Access Manager (see "Get Info" at the end of this Chapter). The process of sending a query, processing it, and returning the results can take a significant amount of time—as long as several minutes or more. Therefore, when your application calls the DAM, be sure to let the user maintain control. The user should be able to cancel a query in progress. This is important, since a query can take a long time to be processed.

If your application can run in the background when processing a query, use the Notification Manager to alert the user of any problems. Do this rather than blasting an alert on top of the current application (which may not be your application). The user can then bring your application to the foreground for more information. Specify the query by name (if possible) in dialog boxes and windows.

If your application supports more than one simultaneous query, provide a list of all current queries and their status for your user. This allows the user to easily understand and control what is happening to all queries.

If your application supports query documents, then you should add a new command, "Get Data...," to the File menu. This command should

bring up the standard file dialog and allow the user to select a query document, which may provide a dialog box for the user to add information about the query.

Apple recommends that your application allow users to limit the amount of disk space used for storing query results. Cancel the query if the amount of space will exceed the limit, if your application can check this in advance. Otherwise, you may want to alert the user when you reach the limit. Users may not know how much data will be retrieved by a query. Allowing users to limit the data storage space gives them control over another aspect of query processing.

▶ Conclusion

In this chapter, you've looked at the Data Access Manager and the Data Access Language. The Data Access Manager provides a generic interface to database management systems and other sources of data. Two sets of routines provide a high-level and a low-level interface.

The Data Access Language provides a powerful generic language for accessing data. You can use this language to access data from a variety of proprietary relational database management systems running on minicomputers and mainframes.

By using the DAM and the DAL, your application can provide users with access to the large amount of existing data. The high-level interface, coupled with query documents, provides this capability for little work on your part. Data-intensive applications will tend to use the low-level interface to provide more power and flexibility.

Get Info ▶

For more information on the Data Access Manager, refer to the Data Access Manager chapter of *Inside Macintosh*, Volume VI. For more information on the Data Access Language, refer to *Data Access Language Programmer's Reference* and *Data Access Language Developer's Guide*, available from APDA.

For more information on SQL, the standard query language for relational databases, refer to *A Guide to the SQL Standard* by C. J. Date (Addison-Wesley, 1988). If you're looking for more information on databases in general, you'll have to decide between two excellent textbooks: *An Introduction to Database Systems* by C. J. Date (two volumes, Addison-Wesley, 1990) and *Principles of Database and Knowledge-Base Systems* by Jeffrey Ullman (two volumes, Computer Science Press, 1988).

11 ▶ The Help Manager

▶ Introduction

The Help Manager provides operating-system support for a consistent style of user-level help. The Help Manager, new with System 7, provides a type of help that answers the question "What is this object?" when the user is pointing at an object on the screen (such as a menu, icon, window title, and so on). This new set of services makes it very easy for a user to interactively explore applications. The Finder supports it, and your applications should too.

The Help Manager is quite visible to users; it appends a Help menu to the right of the menu bar. Figure 11-1 shows the standard Help menu as seen from the Finder. If the user turns Help on, then whenever the cursor is positioned over an object that has help messages defined, a help balloon is displayed. The balloon is positioned so that it does not obscure the object it is describing. The help message can be either a text string, in which case the Help Manager tries to present it in an aesthetically pleasing rectangle, or a picture (PICT), which is displayed directly. When the user moves the cursor outside of the object, the Help Manager removes the help balloon. The services of the Help Manager are modeless so Help is available all the time. The mouse events are not changed; for example, whenever the user holds the mouse button down, the mouse generates a mouse-down event, as always. Help is accessible even from modal dialog boxes. Figure 11-2 shows a typical Balloon Help message from the Finder.

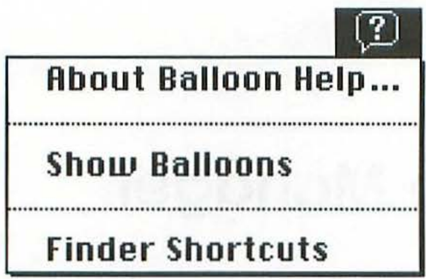


Figure 11-1. The Help menu

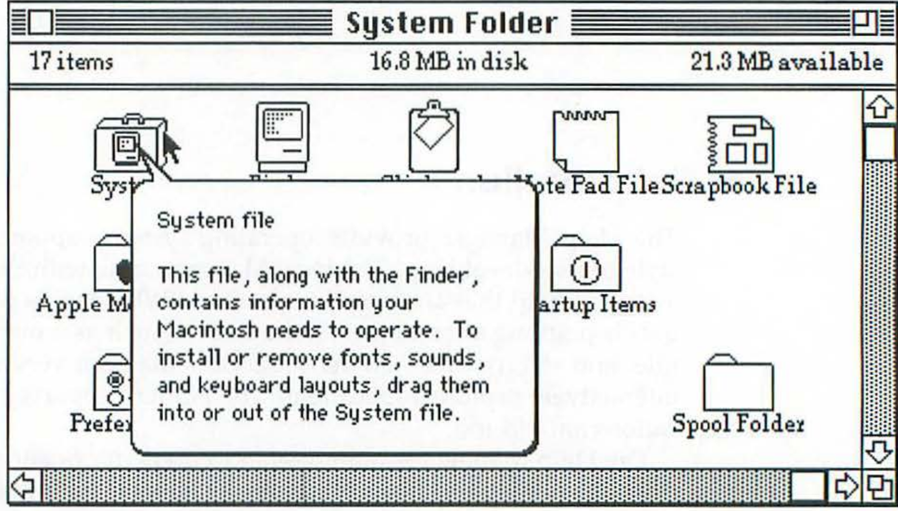


Figure 11-2. A Balloon Help message

The goal of this initial version of the Help Manager is primarily to aid novice users, although more experienced users will also use it occasionally, such as when they're learning an unfamiliar application. The Finder in System 7 uses the Help Manager and provides good examples of how to write help messages.

Note ▶ Other types of help will be provided in future versions of the Macintosh operating system; otherwise, Apple would not have added an entire menu to the menu bar for a single function.

In this chapter, you will look at the facilities of the Help Manager and what is required to add support for this help facility in existing and new applications. You will examine the resources the Help Manager uses as the source of help messages. Last, you'll look at an example of adding help to a particular menu.

▶ Amount of Help versus Amount of Work

System 7 provides a minimal amount of help with no work at all from application developers. If you do nothing at all to support the Help Manager, your users will still find some help available. Standard help messages are provided for the following items:

- Standard window parts (such as the scroll bars and the grow box)
- System dialog boxes (such as the standard file dialog boxes)
- The Apple and Help menus

Help messages are provided for system-supplied control panels. The Finder has help messages for the following:

- All of its menus
- Standard system icons (such as disks and the Trash can)

If you're willing to add resources that provide the help messages to your application, then with no changes to your source code, you can provide help messages for all or almost all of the features of your application. This will enable you to provide help for menu titles, menu items, items in dialog boxes and alerts, and standard window regions. You'll need to write code only if you use a custom menu definition procedure (MDEF) or if you put objects in your windows that users can move around.

For the most part, the help balloon is positioned automatically by the Help Manager. You can specify the position, but the Help Manager will override this if necessary—for example, if the specified position would put the help balloon partially off the screen. The Help Manager uses a WDEF (ID 126) provided as part of the system to draw the help balloon and balloon tip.

The process of writing help messages is considerably simplified if you use BalloonWriter. This utility, which was shipped with every beta version of System 7, allows you to write help messages for standard interface elements, such as menus, windows, and dialog boxes. You can also use it

to create custom balloons, that is, help messages which can only be shown using explicit calls to Help Manager routines.

▶ Compatibility Considerations

If you've added code to support the Help Manager, you'll have to check whether it's running, since the Help Manager didn't exist before System 7. To check whether it is available, use the **Gestalt** system call with the gestaltHelpMgrPresent selector.

If you've only added help resources to your application, then you don't need to check. Earlier versions of the operating system aren't going to know what those resources are anyway so they will be ignored.

Apple recommends that the Help Manager be used to complement existing help systems. If your application supports its own help system, then move any application-specific help menus and menu items to the standard Help menu. Users will come to expect that all help features will be accessible there.

Call the **HMGetHelpMenuHandle** routine to get a copy of the handle to the Help menu. Once you have this handle, use the AppendMenu system call to add your help menu items to this menu. The Help Manager will automatically add a dashed line before any items which you have appended to this menu.

▶ Internationalization Considerations

The help messages in your application will have to be translated if the application is to be used in another language. This includes help messages in PICTs as well as in strings, so your translator will need an application to read and modify the PICT-based help messages. Multilingual applications (applications that can be used in more than one language) are not supported directly by the Help Manager, so each version of an application can support only one language in its help messages at a time. You can write code that calls Help Manager routines to provide multilingual help.

▶ Hot Rectangles!

The Help Manager defines the concept of a *hot rectangle*. When the cursor is positioned over a hot rectangle, then the Help Manager displays the help message defined for it. Notice that it is not a *hot region*—calculations on regions take more time than calculations on rectangles. If the user moves the mouse more than 5 pixels between the time the cursor entered

the hot rectangle and when the Help Manager has calculated that the cursor is over a hot rectangle, the help message is not displayed because the user might be simply moving the mouse somewhere other than the current object.

Some hot rectangles are defined implicitly. Menus that use the system menu definition procedures and items in alerts and dialog boxes are examples of such. Other hot rectangles must be explicitly defined. The hot rectangles for the contents of a window, such as the items on a tool palette, are an example of this.

▶ Help, the State, and Help Messages

Menus, menu items, and many items in dialog boxes can have up to four states:

- enabled
- disabled
- checked-and-enabled
- marked-and-enabled

If an item is marked-and-enabled, then it is enabled and is marked by something other than a check mark or an icon. Some applications, such as Microsoft PowerPoint, use a check mark to show the font of the current selection and a diamond to indicate the default font, if there is no current selection. The Help Manager supports different messages for each of these four states. This provides a more flexible and useful help facility than if the state of objects were not considered.

Help messages can be defined in several different ways:

- 'STR ' resource
- 'STR#' resource
- Pascal string resource
- 'TEXT' and 'styl' resource
- 'PICT' resource

Each set of messages (one message per each of the four states) can use any one of these forms, but all messages in a particular set must use the same form.

The particular form you choose for your help messages depends on the form of the message and on your programming style. If your help

message requires a picture, then you should use a PICT for the help message. If you want text strings, then you can use one of the other formats, depending on your preferences.

▶ Help Resources

The Help Manager chapter of *Inside Macintosh,* Volume VI defines four basic help resources: 'hmnu', 'hdlg', 'hrct', and 'hwin'. We'll look at these four resources and then look at two additional help resources, 'hovr' and 'hfdr'.

Each help resource begins with a header. You can set any of the following flags in the header of a help resource:

- hmUseSubID—use subrange IDs—Set this flag on when the help resource belongs to a desk accessory or some other type of driver that has its own resources.
- hmAbsoluteCoords—Use absolute coordinates in the window—Set this flag if you want the Help Manager to treat the upper left corner of the window as coordinate 0,0 (such as when the port rectangle isn't the same as the window origin).
- hmSaveBitsNoWindow—Save bits, no update event—Set this flag if you want the Help Manager to save the bits behind the help balloon and restore them without generating an update event.
- hmSaveBitsWindow—Save bits, update event—Set this flag if you want the Help Manager to save the bits behind the help balloon, restore them, and then generate an update event. If you use the default, where you set neither the hmSaveBitsNoWindow flag nor the hmSaveBitsWindow flag, the Help Manager will not save the bits behind the balloon, but it does generate an update event when the balloon is removed.
- hwinMatchInTitle—Set this flag if you want the Help Manager to match a window with a 'hwin' resource if the string in the 'hwin' resource is contained somewhere in the window title. The default is that the string in the 'hwin' resource must match the initial characters of the window title.

The default option is that none of these flags is set.

▶ The 'hmnu' Resource

The 'hmnu' resource provides the content of help messages for standard menus. It consists of four sections: a header, a set of messages for missing items, a set of messages for the menu title, and an array of message sets, one per menu item. The structure of the 'hmnu' resource is shown in Figure 11-3. The different kinds of entries for menu items are shown in the lower half of the illustration. Notice that there is an entry type for menu items that should be ignored with respect to help, such as the dotted lines separating menu commands. You can also choose to specify the help message by providing a named resource rather than specifying the messages directly in the 'hmnu' resource. Specifying a set of help messages for menu items that match a string is useful if the menu item can change, but still follows a pattern.

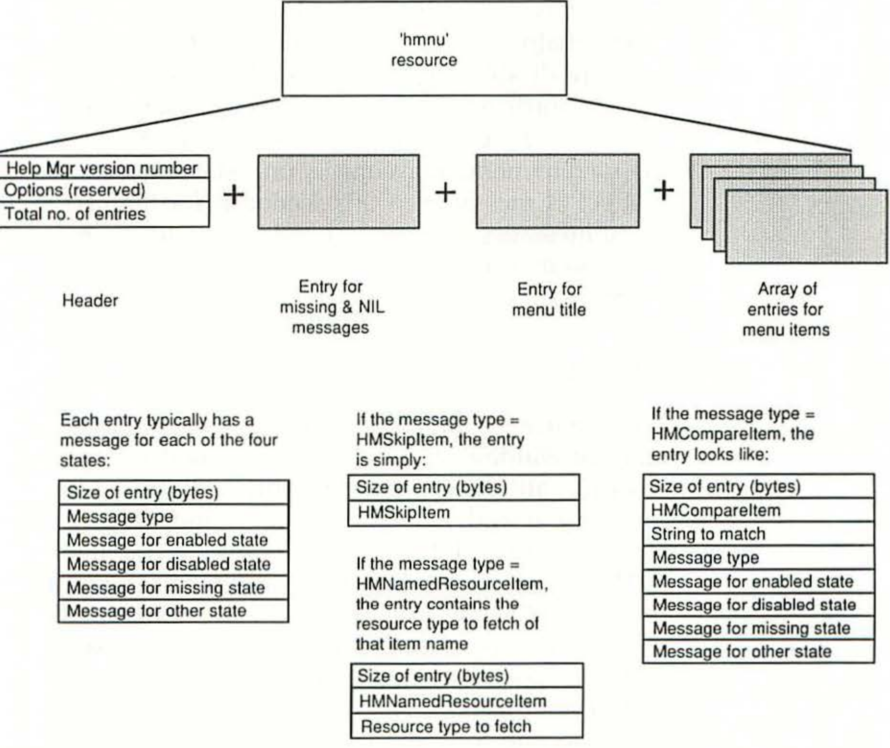


Figure 11-3. The structure of the 'hmnu' resource

The header contains the version of the Help Manager that this resource was developed for, a set of options, the resource ID of the WDEF (Balloon definition function), which will display the help balloons, the preferred position of the help balloons, and a count of the number of entries in the rest of this resource.

The second section contains the set of messages that will be used for any missing items defined later in this resource—one message for each of the four states. If a help message in one of the later sections is defined as "" (the NIL string), then the Help Manager will use the corresponding message in this section in its place. Thus, a message common to many menu items can be defined here and not many times, reducing the amount of space required for the help messages. You can also use this set of messages for missing messages—that is, for items beyond the last menu item for which help messages are provided in this resource.

The third section contains three help messages for the menu title: for the enabled state, the disabled state, and when the menu is disabled by a modal dialog box, and a help message for all menu items to be used when they are disabled by a modal dialog box.

The fourth section contains messages for each menu item in the menu, one set of four messages for each menu item. The sets of messages are mapped to menu items one by one, starting from the top of the menu. The first set of messages in this section is for the first menu item, the second set of messages is for the second item, and so on. A definition is provided the resource description for skipping a menu item. You'd use this, for example, if a menu has dashed lines.

▶ The 'hdlg' Resource

You can use the 'hdlg' resource to hold the help messages for a dialog box, alert, or window. The 'hdlg' resource is structured similarly to the 'hmnu' resource, although it has only three sections: a header, a set of messages for missing and/or NIL messages, and an array of message sets. The structure of the 'hdlg' resource is shown in Figure 11-4.

The header contains the same items as the header for the 'hmnu' resource, plus one additional item: an offset to 'DITL' items. Remember—a 'DITL' resource contains a list of items for a particular dialog. This offset makes it easier for you to provide help when you append one 'DITL' to the end of another 'DITL'.

The second section, the set of messages for missing items, is used in the same fashion as in the 'hmnu' resource.

The third and final section contains a series of messages illustrated in the lower portion of Figure 11-4. Each set of four messages corresponds to an item in a 'DITL'. These are paired one for one; that is, the first set of

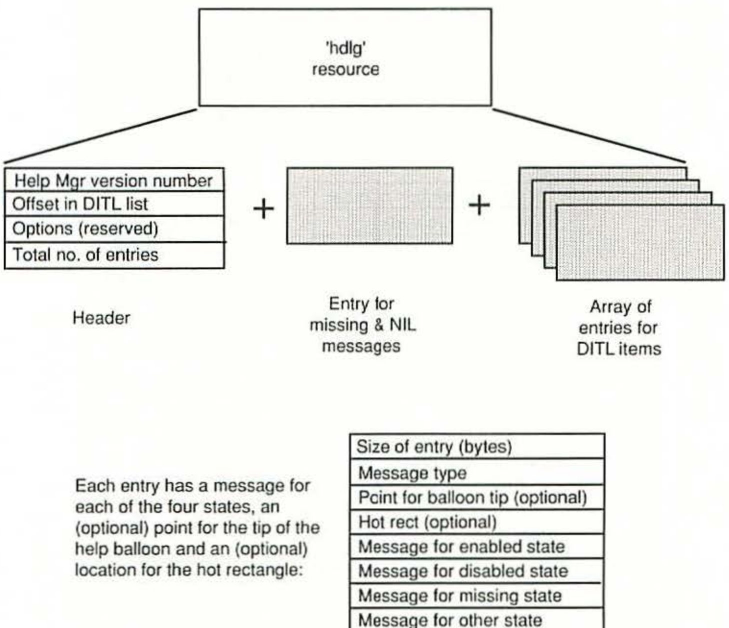


Figure 11-4. The structure of the 'hdlg' resource

messages applies to the first entry in the 'DITL'. The first entry in the 'DITL' is actually the *n*th entry, where *n* equals the offset specified in the header of this 'hdlg' resource. If you aren't satisfied with the default hot rectangle and point, you can specify an optional hot rectangle and point for the balloon tip for each set of messages.

▶ The 'hrct' Resource

The 'hrct' resource is used to specify the hot rectangle and a single help message for a set of objects in a window. Because these resources provide only a single message, unlike a 'hdlg' resource, it does not deal with the state of an object. These resources also have no provisions for NIL or missing messages. The structure of the 'hrct' resource is shown in Figure 11-5.

The 'hrct' resource is structured similarly to the 'hmnu' resource, although it has only two sections: a header and an array of entries. The header contains the same items as the header for the 'hmnu' resource: the version of the Help Manager for which this resource was developed, an area for options, and a count of the number of entries in the rest of this resource.

The second section is an array of entries, one entry per object. Each entry contains a help string, a hot rectangle, and a point for the balloon

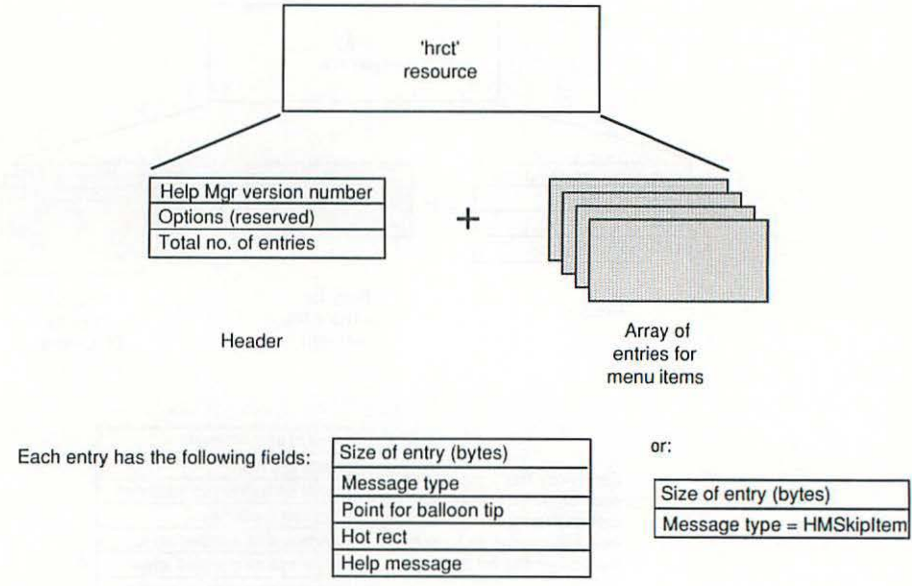


Figure 11-5. The structure of the 'hrct' resource

tip. You must specify the hot rectangle and the point, because the Help Manager has no other source for this information—there are no default values as in menus and items in dialogs. There is also an entry to tell the Help Manager that you don't wish help to be provided for a particular window object.

▶ The 'hwin' Resource

The Help Manager uses the 'hwin' resource to map from window titles to the particular help resource ('hdlg' or 'hrct') to use for that window. At most one 'hwin' resource is needed in a application's resource fork. The structure of the 'hwin' resource is shown in Figure 11-6.

The 'hwin' resource is structured in two sections: a header and an array of information for each set of windows. The header contains the version of the Help Manager that this resource was developed for and an area for options.

The second section is an array of entries, each containing the following:

- A resource ID
- A resource type ('hdlg' or 'hrct')

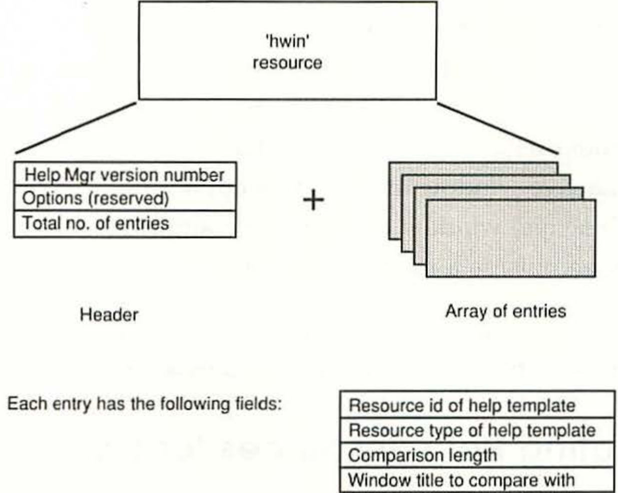


Figure 11-6. The structure of the 'hwin' resource

- A string to compare window titles against
- A comparison length (a match between a window title and the comparison string must be made with at least this many characters)

Add one entry to your application's 'hwin' resource for each window (or set of windows) for which you are providing help.

▶ The 'hfdr' Resource

The Finder has a default help message to display when the cursor is over the icon of an application. You can specify your own help message when the cursor is positioned over the icon of your application. The help message should be in an 'hfdr' resource of ID -5696.

Note that there is no way to override the Finder's default help message for document icons.

▶ The 'hovr' Resource

You can override the default messages for standard interface elements for your application using an 'hovr' resource. In this resource, you can specify a help message for the following elements:

- Title bar of the active window
- Grow box of the active window
- Close box of the active window
- Zoom region of the active window
- Inactive window of the active application
- (Inactive) window of an inactive application
- Area outside a modal dialog box

Apple recommends that you not use this resource unless you have a strong need to override the default messages.

▶ Creating Help Resources for Standard Menus

The 'hmnu' resource just described is used to provide help messages for any standard menu, including menu titles, pull-down menus, hierarchical menus, and pop-up menus.

For each item in a menu, you can write up to four messages, one message for each of the four states: enabled, disabled, checked-and-enabled, or marked-and-enabled. You can provide all four messages for an item, or none. Don't forget that you must have an entry in this resource for dashed lines in the menu as well. A special value tells the Help Manager to skip over this object. By taking advantage of the missing and NIL message feature, you can reduce the size of the 'hmnu' resource and make it a little easier to maintain.

The resource ID of the 'hmnu' resource should be the same ID as the menu for which it is providing help. When the user has the cursor over a menu, the **MenuSelect** call returns the menu's resource ID. If help is enabled, the Help Manager tries to locate an 'hmnu' resource with the same resource ID. If such a resource exists, and there is a help message to display, the Help Manager puts it on the screen.

If the menu can change, the resources get a little more complicated. If the items are simply added to the end of the menu, you can append their help messages to the end of this section. If the resources are added in other places in the menu, then you can use one of two methods to provide help messages. Use the **HMCompareItem** statement in the resource if you want to match a string for the item, or use the **HMNamedResource** statement to provide resources for the item. If you use the latter, you can provide one or more of the four messages for the item. If you provide more than one, you must define them separately.

▶ Creating Help Resources for Modal Dialogs

Under System 7, when a modal dialog box is displayed, the Edit and Help menus are still active. Help can therefore be provided even for modal dialogs.

If the dialog does not have a title, use an 'hdlg' resource to provide its help messages. If the dialog does have a title, then you have a choice in how to provide help. The first choice is to use an 'hdlg' resource as in the previous case. The other choice is to use an 'hwin' resource to connect the window title to the help resource, and either an 'hrct' resource or an 'hdlg' resource to provide the help messages.

You can also specify an 'hwin' resource for a modal or modeless dialog. The Help Manager will use this resource when the cursor is placed over any point in the dialog that does not have an 'hrct' or 'hdlg' resource associated with it. Using an 'hwin' resource in conjunction with other help resources means that you can provide help information for the entire dialog, whether it has dialog items or not.

If you use an 'hrct' resource, you can provide only a single help message per item in the dialog. By using an 'hdlg' resource, you can provide up to four messages for each dialog. If you do use an 'hdlg' resource, you'll also have to add a helpItem entry in the dialog's 'DITL'. This item contains the resource type for help ('hdlg' or 'hrct') and the resource ID.

Whichever of the two resources you use, the first entry in the last section of each resource provides the help message(s) for the first item in the 'DITL', the second item provides the help for the second item in the 'DITL', and so on.

One advantage of using the 'hdlg' resource is that if you append another 'DITL' to the dialog's 'DITL', you can provide the help resources for these additional items by creating another 'hdlg' resource for these additional resources. The second 'hdlg' resource uses the offset in the resource's header to specify where the help messages belong in the dialog box after they are appended. The second 'DITL' must also have a helpItem entry.

▶ Creating Help Resources for Modeless Dialogs and Windows

To create help resources for modeless dialogs and window contents (the system handles help for standard window parts), you need to add an entry to your application's 'hwin' resource for each window and modeless

dialog. In this section, the term *window* means both window and modeless dialog.

To provide the help content for a window, you have the choice of using either an 'hrct' resource or an 'hdlg' resource to provide the help messages. The tradeoff between using these two resources is explained in the previous section.

In order to use an 'hwin' resource, your window *must* have a title. Note that you can give a window a title of nonprintable characters. In such a case the window's title will not be displayed, but you can use an 'hwin' resource to provide help for it.

▶ Creating Help Resources for Custom Menus

If your application uses MDEF (custom menu definition) procedures to provide more complex or graphic menus, then you'll have to modify the MDEF procedure in addition to providing the help resource of your choice. Fortunately, not much work is required to do this. The help messages for custom menus should be provided in an 'hmnu' resource just like any other menu.

An MDEF procedure responds to messages passed to it by the Menu Manager, such as mDrawMsg, mChooseMsg, and mSizeMsg. The change to the MDEF you must make is in handling the mChooseMsg. If help is enabled by the user, then after you draw the highlighted menu item, you need to call **HMShowMenuBalloon**. This causes the Help Manager to display the appropriate help message for this menu item.

Before displaying a help balloon, check that the user has enabled Balloon Help by calling the **HMGetBalloons** routine. This system call returns TRUE if the user has enabled Balloon Help. Your application can also enable or disable Balloon Help by means of the **HMSetBalloons** system call (although there are very few situations in which you might need to do this).

When the menu item is unhighlighted, you need to call **HMRemoveBalloon**, which causes the Help Manager to remove the help balloon displayed for this item.

If you want to do some additional work, you can specify a WDEF (window definition) procedure, which will be called by the Help Manager to display a custom balloon shape. You can also write custom functions that calculate the position of the balloon tip and the dimensions of the help balloon.

▶ Creating Help Resources for Movable Window Objects

Providing help for movable window objects is the most complex case—you'll actually have to write some code to do this. This code needs to be written to run at idle time. Basically, the code has to check whether the cursor is over a hot rectangle. This is not so easy, because the objects you're dealing with can move around. If the cursor is over a hot rectangle, and help is enabled, then you need to call **HMShowBalloon**. Don't call **HMShowBalloon** if this rectangle is the same as the last hot rectangle, or your help messages will flash on the screen. As with the **HMShowMenuBalloon** call, you can optionally specify a WDEF procedure that will display a custom help balloon and create functions to calculate the position of the balloon tip and the dimensions of the help balloon.

You should call **HMRemoveBalloon** when the cursor is inside your frontmost window, but not in a hot rectangle. If your windows are resizable, the window contents can change (such as a scrollable palette of tools), or hot rectangles in the content region can change position, then you will need to use the **HMSShowBalloon** system call if you want to provide help.

▶ Using the Help Manager Routines

In addition to the routines discussed earlier in this chapter, the Help Manager provides several other routines that are useful to application programmers. Let's briefly review them.

Call **HMIsBalloon** to check whether a help balloon is currently being displayed on the screen. This routine returns TRUE or FALSE.

Call **HMGetFront** and **HMGetFontSize** to get the current font and font size, respectively, used to display help messages in your application. You can change these values by using the **HMSetFont** and **HMSetFontSize** system calls. Note that these calls are associated only with help messages specified by 'STR ', 'STR#', and Pascal string resources. They have no effect on messages specified by 'PICT' or 'TEXT' and 'styl' resources.

Call **HMSetDialogResID** to change the 'hdlg' resource ID for the next dialog box to be displayed. To find out the 'hdlg' resource ID of the next dialog box to be displayed, call **HMGetDialogResID**. This routine returns −1 if the 'hdlg' resource ID was not set by means of the **HMSetDialogResID** call, or an 'hdlg' resource ID if that routine was used.

Similarly, call **HMSetMenuResID** to change the 'hmnu' resource ID for a menu. You can use this call to override an existing 'hmnu' resource or to

set the ID for a menu for which no 'hmnu' was specified. To find out the 'hmnu' resource ID of the next dialog box to be displayed, call **HMGetMenuResID**. This routine returns −1 if the 'hmnu' resource ID was not set by means of the **HMSetMenuResID** call, or an 'hmnu' resource ID if that routine was used.

To search an open resource file for an 'hdlg' or 'hrct' resource and apply it to the frontmost window, call **HMScanTemplateItems**. Specify the resource type, resource ID, and which resource file should be searched.

▶ Example: Creating an 'hmnu' Resource

Let's now look at how to create a help resource for a particular menu—the Mark menu of MPW (Macintosh Programmer's Workshop), as shown in Figure 11-7. The version of the menu shown on the left of Figure 11-7 has no marks in the current window. The menu on the right has two marks in the current window. *Marking* a selection is like placing a bookmark in a book—it names a particular location in a file. Typical objects that are marked in MPW are function and procedure headers. The Mark menu lists all the marks in the active window below the two commands in the Mark menu. Selecting one of these marks scrolls that selection into view.

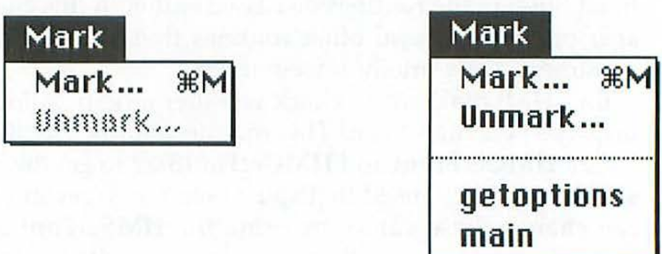


Figure 11-7. MPW's Mark menu

To construct a help resource for this menu, you have to construct an 'hmnu' resource, as shown in Figure 11-8. The resource ID for this resource is 133, because that is the resource ID of the 'MENU' resource for the Mark menu.

The header shows that there are five entries in the 'hmnu' structure. The first entry, for missing and NIL entries, holds the help messages for the marks made in a window by the user. Since you don't know how many

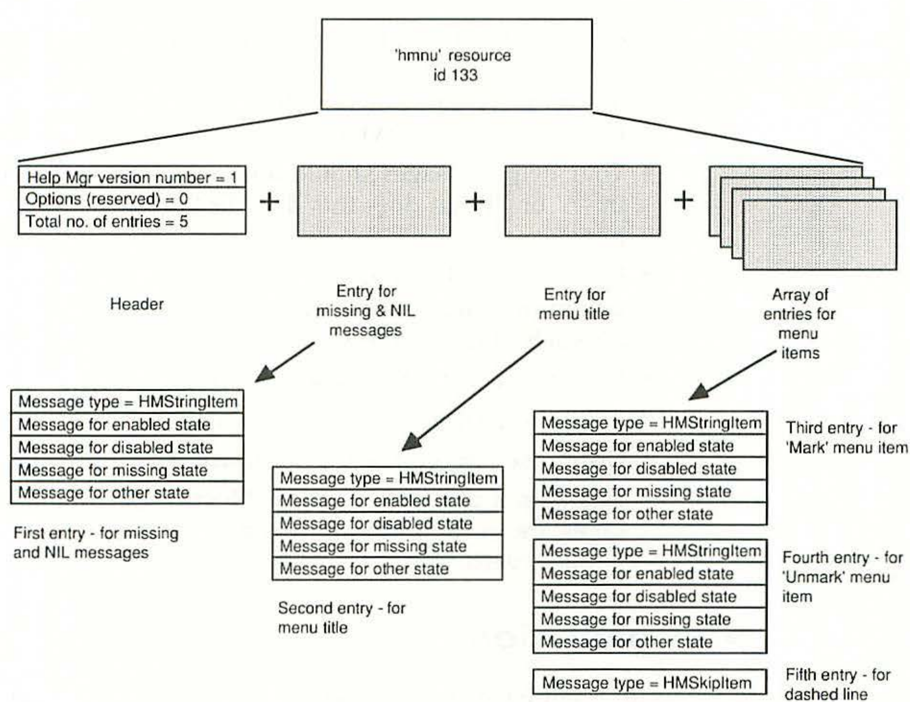


Figure 11-8. The structure of an 'hmnu' resource for the Mark menu

marks the user might create, use the missing message facility in this resource.

The second entry holds the help messages for the menu title.

The remainder of the entries are for the menu items in the Mark menu. The third and fourth entries are for the Mark and Unmark menu items. The fifth and last entry is for the dashed line that separates the commands in the menu from the names of the marks. The Help Manager will not use this entry when there are no marks, because this item is added to the menu after the user has created one or more marks in the current window.

▶ Writing Help Messages

Apple has disseminated a set of rules for writing help messages. These rules have been tested by Apple against real users, and by following them, your users will get more from using the help facility than they might otherwise.

The rules for writing help messages, briefly stated, are as follows.

- Be complete and concise—long help messages may confuse users. Don't repeat the obvious.
- Your help messages should answer questions such as "What is this?" or "What does this do?" or "What happens when I click here?"
- For a set of related objects, such as a set of radio buttons, provide a help message for the set. You don't need to provide a message for each object in the set.
- Name objects that the user will have to learn.
- State the goal first, and use the active voice.
- Don't use jargon; use nontechnical terms and standard phrases.
- Maintain a consistent syntax. Most of the system-related help messages are in the form of "X is a . . . ," "goal + action," and "item name + verb."

▶ Conclusion

In this chapter, you've looked at the new Help Manager. The Help Manager in System 7 answers the question "What is this thing?" when the user moves the mouse over an object on the screen. For the most part, you can provide this kind of help by adding some new resources to your application. However, in a couple of cases you'll have to add code to support the Help Manager.

Get Info ▶

For more information on the Help Manager, refer to the Help Manager chapter of *Inside Macintosh*, Volume VI.

When writing help messages, you may want to refer to an APDA publication, *Apple Publications Style Guide*, which lists much of the standard Macintosh terminology.

12 ▶ The Sound Manager

▶ Introduction

The Sound Manager provides several methods for creating and playing sounds on the Macintosh. The latest version of the Sound Manager adds several new capabilities, which you'll examine in this chapter. You should understand something about the Sound Manager if you use *any* sounds in your application, including the **SysBeep** call.

The Sound Manager in System 7 brings several new sound capabilities:

- Plays sounds continuously from disk while your application and other applications are running
- Plays multiple channels of sampled sound at a time and has them mixed in real time
- Monitors the CPU load that one or all sound channels are using
- Monitors the status of sound channels
- Records sounds

If your application does not deal primarily with sound or music, then you will not need to understand the Sound Manager in detail. You can play short and long sounds by using a couple of system calls. If your application is a music application, then you'll have to understand everything about the Sound Manager.

In this chapter, you'll first look at the three different ways of producing sounds on the Macintosh and the data structures used by the Sound

Manager to do this. Following that, you'll go through the high-level calls to the Sound Manager. Then you'll look at architecture of the Sound Manager and the sound commands used to create and manage sounds. You'll then look at the low-level interface to the Sound Manager. Last, you'll look at how to record sounds on the Macintosh, first using the high-level interface and then the low-level interface.

▶ Different Ways to Produce Sounds

The Sound Manager supports three different methods of creating sound. The simplest method synthesizes notes, one at a time. You specify a note by describing its frequency, volume, and duration. This method of creating sound can be used only to create simple melodies and perhaps some sound effects. It is not suitable for producing speech.

The second method requires that you specify up to four wave tables. A *wave table* describes a single wave of sound as a series of numbers. When the Sound Manager plays a sound described in a wave table, it sends the numbers in the wave table out one at a time to a digital-to-analog converter. When the last number has been sent out, the Sound Manager returns to the beginning of the wave table and begins all over again. This method produces higher-quality sound than the first method, but it is not suitable for speech and complex sounds without a great deal of work on your part. Figure 12-1 illustrates how a wave table is used by the Sound Manager to produce sound.

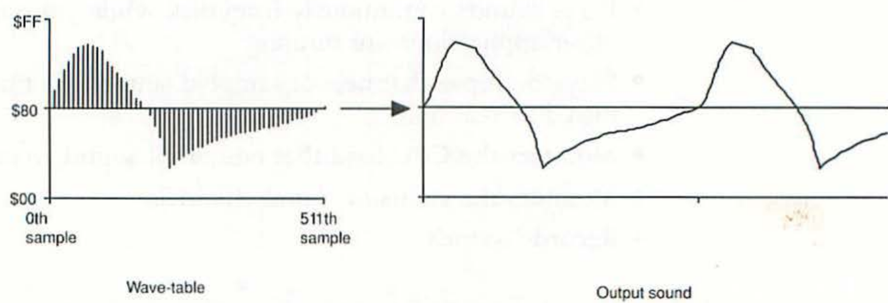


Figure 12-1. How the Sound Manager uses a wave table

The third method requires that you have digitally recorded the sound you want to play. You can do this using the sound input capabilities of the Macintosh LC or IIsi, or the Farallon Sound Recorder, for example. In this method, you provide all the data for the sound—similar to the way sound

is recorded on CDs. This third method is similar to the second method except that instead of describing a periodic waveform, you are describing the entire sound from start to finish. You can use this method to produce speech and virtually any other sound.

The Sound Manager uses a different 'snth' resource, or synthesizer, to take each of these kinds of data and output sound. Synthesizers are described later in this chapter.

▶ Sound Data Structures

The Sound Manager can use either an 'snd ' resource or a file as a source from which to create sounds. You can use 'snd ' resources to describe sounds, digitized voice, special effects, and so on. The file format for digitized sound is especially useful for storing large amounts of sound data. You will learn about the resource and the file format briefly, and then look at how they are used to create sounds. You will later revisit them to examine their detailed internal structure.

▶ Sound Resources

You can use the 'snd ' resource to describe how to create a sound by using sound commands, which will be described in a later section, and optional sound data. An 'snd ' resource specifies which synthesizers should be used to create sound: note, wave table, or sampled sound.

The 'snd ' resource has two different formats. Format version 1 is the general format and is the only format that application programmers should use. Format version 2 is specifically for HyperCard and should not be used, since version 1 is a superset of version 2. The general structure of an 'snd ' resource is illustrated in Figure 12-2. In the figure, note that the arrays are variable in length.

Note ▶ Sound resource IDs 0 to $1FFF are reserved for Apple. Resource IDs 1 to 4 are defined to be the standard system beeps.

▶ Sound Files

Sounds can be stored in a disk file using the Audio Interchange File Format (AIFF). This format is not described in *Inside Macintosh*, but in a separate document available from APDA.

'snd ' format version
Number of synthesizers
Array of synthesizers
Number of sound commands
Array of sound commands
Offset to data
Number of sound samples
Sampling rate
Offset to start of loop
Offset to end of loop
Base note
Array of sound data

Each element in the array of synthesizers looks like:

Synthesizer resource id
Initialization parameter

Each sound command in the array of commands looks like:

Command
Parameter 1
Parameter 2

The array of sound data is simply a string of bytes (one byte per sample).

Figure 12-2. The structure of a format version 1 'snd ' resource

▶ Calling the Sound Manager at a High Level

The high-level calls to the Sound Manager—**SysBeep**, **SndPlay**, and **SndStartFilePlay**—provide easy, but still powerful ways to use sound in applications. Some calls can be used as either a high-level or a low-level call. These calls are discussed in this and a later section. If you need only high-level sound capability, you'll only need to read this section to be able to use the Sound Manager.

▶ Calling SysBeep

The simplest way to produce sound is with the **SysBeep** call, which was moved from the System Utilities to the Sound Manager in System 7. You must supply a parameter when calling **SysBeep** for compatibility reasons, but the Sound Manager ignores it. The Sound Manager tries to reserve enough CPU time to play the beep if the system beep is enabled.

By the Way ▶ If you play the simple beep on a Macintosh Plus or SE, the Sound Manager will use the ROM code instead of the new Sound Manager code. This is to maintain compatibility with some early third-party MIDI applications.

Enable and disable the system beep by calling the **SndSetSysBeepState** routine. **SndGetSysBeepState** returns the current state of the system beep. The default state is enabled.

Important ▶ Consider using the Notification Manager if you're using **SysBeep** to warn the user. The Notification Manager offers more features for not much more work.

▶ Producing Sounds the Easy Way with SndPlay

The most painless way to produce interesting sounds is with the **SndPlay** call. All you have to do is pass it the handle to an 'snd ' resource, and you've got sound. You don't have to know anything about modifiers, synthesizers, or channels. **SndPlay** can play sounds from sound commands and/or sound data. The sound data can be compressed or uncompressed.

▶ Using SndStartFilePlay to Play Sounds on Disk

SndStartFilePlay allows you to play arbitrarily long sounds from either a disk file or a resource, and it turns your Macintosh into an expensive tape recorder. If you use **SndStartFilePlay** as a high-level call, then the Sound Manager automatically allocates and deallocates a sound channel. This call must be executed synchronously as a high-level call, because you have no way to be sure that the call has completed when quitting the application.

▶ Architecture of the Sound Manager

The architecture of the Sound Manager allows for both simple and complex uses of sound. An overview of the architecture is shown in Figure 12-3. The major components of the architecture include application(s), the Sound Manager, sound channels, modifiers, synthesizers, and sound hardware. Let's look at each of these components in turn.

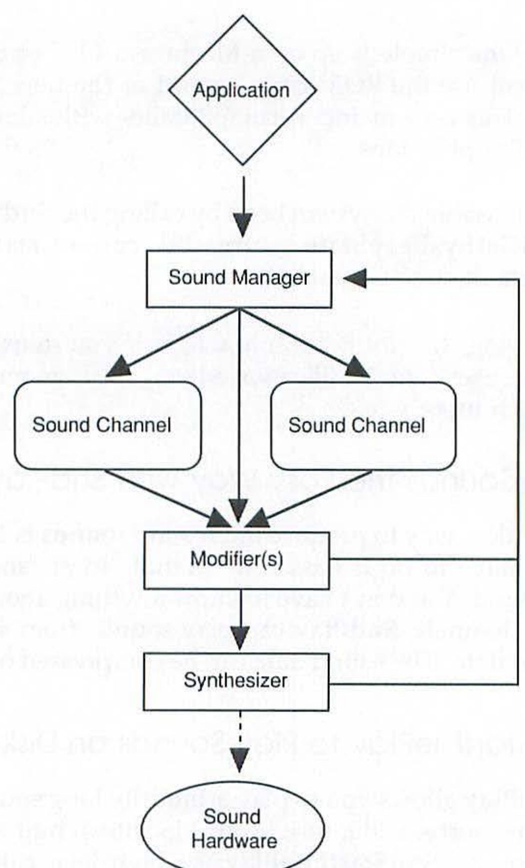


Figure 12-3. The architecture of the Sound Manager

▶ Applications and Sound Channels

Applications create sound commands by making calls to the Sound Manager. An application can have more than one channel open at a time, and several applications can have channels open at the same time.

A sound channel is a first-in, first-out (FIFO) queue of sound commands. Each command channel is owned and operated by one application and channels cannot be shared between applications. A sound channel is linked to a particular synthesizer. Most applications do not have to deal with sound channels directly because they are managed by the high-level calls of the Sound Manager.

▶ Modifiers

Modifiers are small pieces of code that filter or process sound commands before they reach their intended synthesizer. You might create a modifier to change the key of all notes passing through it, or to double the length of each note. Since modifiers operate in real time, they cannot perform complex tables which take a long time to complete. More than one modifier can be installed into a channel, although modifiers can only be installed into channels opened with the **SndNewChannel** call.

The only communication between a modifier and the Sound Manager is the return value of the modifier. If the return value is FALSE, then the modifier has completed processing a command. If the return value is TRUE, then the modifier is still processing the command.

A modifier can change, remove, or ignore commands passed to it. Also, a modifier can create new commands that will be processed by any modifiers that follow it in the queue or the synthesizer. A modifier can send commands to its succeeding modifier, the synthesizer (if there are no more modifiers), or the Sound Manager.

A modifier must support the **initCmd** and **freeCmd** commands, which the Sound Manager uses to install and remove modifiers. Your application can also send commands to the first modifier in a chain, bypassing the command queue. You might do this to stop all sound immediately. Note that your application cannot send commands directly to any succeeding modifiers.

▶ Synthesizers and 'snth' Resources

Synthesizers, or 'snth' resources, are used to talk to the sound hardware. These resources are similar in nature to device drivers, but they have a different interface. Table 12-1 lists the various synthesizer resource IDs and their uses. The two kinds of synthesizers are as follows:

- Playback synthesizers, which control hardware to create sounds
- Utility synthesizers, which are used to compress and decompress sound data

Synthesizers can send commands back to the Sound Manager.

To request the playback synthesizers, use the generic resource ID—that is, the ID should be in the range 0 through $799. The Sound Manager is responsible for choosing the appropriate synthesizer for the hardware on which your application is running. If the application is running on a Macintosh Plus or SE, the Sound Manager will add $1000 to the requested

Table 12-1. The 'snth' resources and their uses

Resource ID (hex)	*Name*	*Type*	*Supported Macintosh Computers*
1	Note synthesizer	playback	All
3	Wave-table synthesizer	playback	All
5	Sampled-sound synthesizer	playback	All
11	MACE (3:1)	utility	All
13	MACE (6:1)	utility	All
0–FF	Reserved for Apple	—	All
100–799	Available for developers	—	All
801	Note synthesizer	playback	Mac with ASC
803	Wave-table synthesizer	playback	Mac with ASC
805	Sampled-sound synthesizer	playback	Mac with ASC
800–8FF	Reserved for Apple	—	Mac with ASC
900–999	Available for developers	—	Mac with ASC
1001	Note synthesizer	playback	Mac Plus or SE
1003	Wave-table synthesizer	playback	Mac Plus or SE
1005	Sampled-sound synthesizer	playback	Mac Plus or SE
1000–10FF	Reserved for Apple	—	Mac Plus or SE
1100–1199	Available for developers	—	Mac Plus or SE

synthesizer resource ID. On the other hand, if your application is running on one of the other machines that has an ASC (Apple Sound Chip), the Sound Manager adds $800 to the requested resource ID. If you requested the note synthesizer (ID 1), the Sound Manager would actually call 'snth' ID $1001. If you were running on a Macintosh IIcx, you would actually be using 'snth' ID $801. Software that requests a synthesizer by its generic ID will therefore work with third-party hardware and with any future Macintoshes that implement sound using different hardware.

The MACE synthesizers, which are built into system software under System 7, implement a software-only compression and decompression scheme. MACE stands for Macintosh Audio Compression and Expansion, and these routines are separately available as a product through APDA. You'd need the APDA version of MACE only if you were planning on supporting System 6. These routines are needed because sound data grows big very quickly. You can easily accumulate a megabyte of data per minute of recorded uncompressed sound. The MACE synthesizers pro-

vide either 3:1 or 6:1 compression and expansion. The 3:1 compression maintains reasonably high quality, and the 6:1 compression provides lower-quality sound, but higher savings in storage space.

Files that have been compressed using this code can be expanded and played in real time on all Macintosh computers using the sampled-sound synthesizers. Only the right channel of stereo data is expanded and played on the Macintosh Plus, SE, and Portable. On Macintosh IIs, you can also compress sound in real time and expand and play back in stereo.

You can attach only one synthesizer to a given piece of sound hardware. You can attach more than one channel to a synthesizer.

Synthesizers of ID 1, 3, 5, 11, 13, 801, 803, 805, 1001, 1003, and 1005 are provided by System 7. The first three synthesizers (1, 3, and 5) are placeholders—no code is directly associated with them. By using these three synthesizers, the Sound Manager, rather than your application, decides at runtime whether to use the $800 or $1000 series of synthesizers.

▶ Sound Hardware

One of the primary goals of the Sound Manager is to hide the details of the hardware from application developers. Applications should always talk to the Sound Manager and never directly to the hardware. This is important because the standard sound hardware on Macintosh computers was improved between the Macintosh SE and the Macintosh II family of computers. If an application was talking directly to the sound hardware on a Macintosh SE, it would not have worked on a Macintosh II. In addition to Apple sound hardware, third-party sound hardware is available.

The sound hardware is the lowest level of the sound architecture. This hardware translates commands and data into sound, generally using a digital-to-analog converter to do the job. A Sony chip is used in the Macintosh Plus and SE. The Apple Sound Chip (ASC) is used on all later models.

▶ Limitations

The number of channels that can be open at one time is limited primarily by the CPU speed and the sound hardware. A Macintosh II can support several simultaneous channels, but a Macintosh Plus has enough CPU power to support only one channel. An ASC is required to support multiple channels. Another limitation is that only the sampled-sound synthesizer allows you to mix multiple channels of sound.

▶ Sound Commands

You use commands to control the process of creating sound. Commands can be issued by an application, the Sound Manager, modifiers, and synthesizers. Applications can issue commands one at a time or many at a time by using an 'snd ' resource. Commands can be issued to the Sound Manager, modifiers, and synthesizers.

Sound commands have a single structure: 2 bytes for the command type, 2 bytes for the first parameter, and 4 bytes for the second parameter. The parameters are used for different purposes by different commands. Table 12-2 lists all the Sound Manager commands, who might issue them, and what their purpose is.

The high-order bit of the command field is used as a Boolean. This field is used only if the command is in an 'snd ' resource. If the bit is set, then the second parameter specifies the offset from the beginning of the resource to where the data is located.

Table 12-2. Sound commands sorted by command type and command name

Command (2 bytes)	*Parameter 1 (2 bytes)*	*Parameter 2 (4 bytes)*	*Issuer*	*Command Type*	*Description*
timbreCmd	timbre	0	A, M	n	Changes timbre of a sound
ampCmd	amplitude	0	A, M	n, w, s	Changes loudness of a sound
freqCmd	0	frequency	A, M	n, w, s	Changes pitch of a note
noteCmd	duration	amplitude and frequency	A, M	n, w, s	Plays a note
restCmd	duration	0	A, M	n, w, s	Rests a channel
waveTableCmd	length	pointer	A	w	Installs a wave table
bufferCmd	0	pointer	A, S	s	Plays a sampled sound
continueCmd	0	pointer	A	s	Continues playing a sampled sound
convertCmd	compression ID	pointer	A	s	Compresses or expands sound data
rateCmd	0	rate	A	s	Changes the pitch of a sampled sound
reInitCmd	0	initialization options	A, S	s	Reinitializes a channel

Table 12-2. Sound commands (continued)

Command (2 bytes)	*Parameter 1 (2 bytes)*	*Parameter 2 (4 bytes)*	*Issuer*	*Command Type*	*Description*
sizeCmd	compression ID	pointer	A	s	Changes the pitch of a sampled sound
soundCmd	0	pointer	A	s	Installs a sound as an instrument
callBackCmd	application defined	application defined	A	c	Executes a callback procedure
emptyCmd	0	0	S	c	Does nothing
pauseCmd	0	0	A, M	c	Pauses processing
resumeCmd	0	0	A, M	c	Resumes processing
syncCmd	count	identifier	A	c	Synchronizes channels
waitCmd	duration	0	A, M	c	Suspends processing in a channel
availableCmd	result	initialization options	A	u	Checks whether an initialization is supported
flushCmd	0	0	A	u	Flushes a channel
freeCmd	0	0	S	u	Frees a channel
howOftenCmd	period	pointer	M	u	Sets period for tickleCmd
initCmd	0	initialization options	S	u	Initializes a channel
loadCmd	0	initialization options	A	u	Reports CPU load
nullCmd	0	0	M	u	Does nothing
quietCmd	0	0	A	u	Stops a sound
requestNextCmd	count	0	S	u	Sends next command
tickleCmd	0	0	S	u	Does a periodic action
totalLoadCmd	0	initialization options	A	u	Reports CPU load
versionCmd	0	version	A, S	u	Reports synthesizer version
wakeUpCmd	period	pointer	M	u	Sends a tickleCmd

Codes for Issuer: A = application, M = modifier, S = Sound Manager

Codes for Command Type: n = note synthesizer command, w = wave-table synthesizer command, s = sampled-sound synthesizer command, u = utility command, c = synchronization-control command

Some of these sound commands will be explained in the following sections. Refer to the Sound Manager chapter of *Inside Macintosh*, Volume VI for the details of the rest of them.

▶ Calling the Sound Manager at a Low Level

The low-level calls to the Sound Manager give you more control over sound, at the expense of additional complexity in your code.

▶ Managing Channels

To use the low-level calls, first allocate and open a sound channel with the **SndNewChannel** call. At this time, you'll tell the Sound Manager which synthesizer to use and how to initialize it. For wave-table synthesizers, you can ask for one of four channels. For sampled-sound synthesizers, you can ask for the left and/or right channel, mono or stereo output, handling of 3:1 or 6:1 compressed sound data, and the sampling rate. You can also specify some conversion options. The default size for a channel is 128 commands.

When you are finished, close and deallocate the channel using the **SndDisposeChannel** call. When making this call, specify whether the channel should be flushed before closing it. Close the channel as soon as you're done with it. The Sound Manager attempts to manage CPU cycles, and it will not release the cycles associated with a channel until that channel has been closed.

▶ SndDoCommand and SndDoImmediate

A more powerful but more complex way to use sound is with the **SndDoCommand** and **SndDoImmediate** calls. Once a channel has been successfully opened, you can use **SndDoCommand**, which places a single sound command in the sound channel you specify. You can also indicate whether the Sound Manager should return with an error if the queue is full, or wait until there is space to add it.

The **SndDoImmediate** call directly places a single sound command to the first modifier or the synthesizer (if there are no modifiers), and it bypasses the command queue maintained by the channel.

▶ Playing Notes and Installing Instruments

Use the **SndDoCommand** or **SndDoImmediate** call and a **noteCmd** sound command to play individual notes on a channel. You'll need to specify the duration, volume, and frequency of the note in the command. Notes can be played using any of the three standard synthesizers.

Use the **waveTableCmd** sound command to install a wave table as an instrument. The first parameter for this command is the length of the wave table, and the second parameter is a pointer to the wave table. If the wave table is not 512 bytes long, the Sound Manager resamples it so that it becomes 512 bytes long. You can then play notes using the wave table synthesizer by using **noteCmd** sound commands.

Use the **soundCmd** sound command to install a sampled sound as an instrument. The first parameter should be NIL. The second parameter is a pointer to the sampled sound (which should be locked in memory), which can be an 'snd ' resource. In this case, pass a pointer to this resource in **soundCmd**. When you're done with the sound, unlock it.

▶ Playing Sampled Sounds

Use the **SndDoCommand** or **SndDoImmediate** call to play and control sampled sounds. If you want to play a sound only once, you can use the **bufferCmd** sound command. The sound can be compressed or not. The first parameter should be NIL. The second parameter for this command is a pointer to a standard sound header (if the sound is mono and is not compressed), an extended sound header (for stereo sounds), or a compressed sound header. These headers are described in detail in the Sound Manager chapter of *Inside Macintosh*, Volume VI.

The **rateCmd** sound command allows you to change the pitch of the sound, and the **continueCmd** sound command enables you to play a long sampled sound in smaller sections. You should use **bufferCmd** for the first section and **continueCmd** for the following sections.

▶ Playing Sampled Sounds from Disk

You can control the process of playing sampled sounds from disk by calling lower levels of the Sound Manager. To do this, call **SndNewChannel** to first open a channel. Then you can call **SndStartFilePlay** to play either an 'snd ' resource from disk or an AIFF-formatted file of sound data. At this time, specify whether the sound should be played synchronously or asynchronously.

If you choose asynchronous play, you can control the channel by using the **SndPauseFilePlay** and **SndStopFilePlay** system calls. These calls pause or stop the sound channel that is playing the sound.

▶ Managing Sound Channels

You use the **quietCmd**, **flushCmd**, **waitCmd**, and **pauseCmd** sound commands to control the activity of a sound channel.

The **quietCmd** sound command stops a sound in progress if it is issued by a **SndDoImmediate** call. This sound command stops the sound being played, and the channel continues to the next command in the queue.

The **flushCmd** sound command removes all the sound commands, but will not disturb the sound currently being played. The Sound Manager sends a **flushCmd** followed by a **quietCmd** when you call **SndDispose-Channel**. This is what you should do as well if you want to completely stop a channel.

Use the **waitCmd** sound command to suspend all activities on a channel for the specified amount of time. This command is useful with both the **SndDoCommand** and **SndDoImmediate** calls. In the former case, the channel will be suspended whenever **waitCmd** becomes the current command. In the latter case, the channel will be suspended immediately.

To suspend all activities on a channel for an indefinite amount of time, use the **pauseCmd** sound command. Nothing further will happen on this channel until it receives a **tickleCmd** or **resumeCmd** sound command.

By adding a **syncCmd** sound command to each channel, you can synchronize more than one sound channel. The parameters for this command are an identifier and a count. Activity on a channel is suspended when **syncCmd** becomes the current command. Activity is resumed when each **syncCmd** associated with a particular identifier becomes the current command. The count tells the Sound Manager how many channels are to be synchronized.

▶ Managing Channel Capacity

Sound, especially high-quality sound, requires system resources. The major determinant of the number and quality of channels that can be supported at a time is the CPU capacity. The Sound Manager provides two sound commands that can help you decide the number and quality of channels to use at runtime.

The **totalLoadCmd** sound command returns the total percentage CPU load that a new channel with the specified initialization options and any existing open channels would use. The Sound Manager figures that if no

channels are open, 0 percent of the CPU capacity is in use. This is an approximation since the Sound Manager has no way of knowing what low-level system code (network drivers and so on) might be doing. If the CPU load returned by **totalLoadCmd** is greater than or equal to 100 percent, then you probably don't want to open that new channel.

The **loadCmd** sound command returns the percentage CPU load that a new channel with the specified initialization options would use.

▶ Compressing and Expanding Sound

If you want to compress or expand sound data, then you'll need to use the **convertCmd** sound command. If you need to play a compressed sound, you wouldn't use these sound commands, but rather the **SndPlay** call or the **bufferCmd** sound command. The **convertCmd** sound command uses the MACE utility synthesizers.

The **sizeCmd** sound command is useful if you need to know the amount of space that the converted data will require. You'd need to know this if you were managing the memory required for the conversion process.

To compress sound data, issue the **convertCmd** sound command using the **SndControl** system call. The parameters for this call are the synthesizer resource ID (to select 3:1 or 6:1 compression ratio) and a sound command. The parameters for the sound command are the compression ratio and a pointer to a conversion block. A conversion block contains pointers to a source and a destination compressed sound header.

To expand compressed sound data, issue the **convertCmd** sound command using the **SndControl** system call. The parameters for this call are the synthesizer resource ID (to select 3:1 or 6:1 compression ratio) and a sound command. The parameter is a pointer to a conversion block, which contains pointers to a source and a destination compressed sound header. These headers tell which compression ratio is to be used.

There are also some high-level routines for compressing and expanding sound data at ratios of 1:3 or 1:6. These routines are named **Comp3to1**, **Comp6to1**, **Exp1to3** and **Exp1to6** and were previously available as part of the MACE package. All four of these routines work in memory; that is, they expand or compress sound data from one memory buffer to another. They cannot work directly with sound data in disk files.

▶ Compatibility and the Sound Manager

At this time, you cannot determine whether the Sound Manager on a machine can perform certain functions unless you know whether the machine has an ASC.

Note ▶ In future versions, you will be able to check for specific sound capabilities through additional calls to **Gestalt**. Until this happens, you'll have to check for the existence of the chip.

Use the **Gestalt** system call with a selector of gestaltHardwareAttr to determine whether the current machine has an ASC. If the gestaltHasASC bit is set, then the machine has the chip.

Call **Gestalt** with the gestaltSoundAttr selector to determine whether the current machine can play stereo sounds (in stereo, not in mono) or whether you can mix the channels on the external speaker. If the gestaltStereoCapability or the gestaltStereoMixing bit is on, then those capabilities are available.

You will also be informed if the sound input routines are available, if any built-in input device is available, and if any sound input device is available.

To find out which version of the Sound Manager is running on the current machine, call **SndSoundManagerVersion**. This call returns a long integer whose contents should be interpreted as the first 4 bytes of a 'vers' resource.

If you are using the MACE routines, check which version is available by calling **MACEVersion**. This call also returns a long integer whose contents should be interpreted as the first 4 bytes of a 'vers' resource.

If your application is using the sound input capabilities of the Sound Manager, verify which version is running by calling **SPBVersion**. It also returns a long integer whose contents should be interpreted as the first 4 bytes of a 'vers' resource.

▶ Getting Sound Status Information

To find out the status of a sound channel, use the **SndChannelStatus** call, which returns the initialization parameters that were accepted and the CPU load for this channel. If the channel is being used to play a sound from disk, this call returns whether the channel is paused or busy. If this channel is busy, this call also returns the starting time, ending time, and current time for playing from disk.

To inquire about the status of all sound channels that have been opened by all current applications, call **SndManagerStatus**. This call returns the maximum load on all channels, the number of allocated channels, and the current load on all channels.

▶ Installing Modifiers

You can install a modifier by calling **SndAddModifier**. To specify the modifier, you use either a procedure pointer or an 'snth' resource ID. You need to specify the channel (which must be open), and you can optionally specify some initialization parameters, which are unique for each modifier. For the details on how to write a modifier, refer to the Sound Manager chapter of *Inside Macintosh*, Volume VI.

The modifier is installed ahead of any previously installed modifiers and the synthesizer. Modifiers can be installed only on channels that were opened with a **SndNewChannel** call.

▶ Recording Sounds

Sound input was added to the Macintosh operating system with the introduction of the Macintosh LC and IIsi, just before the introduction of System 7. Let's look at how to use this new capability.

▶ Sound Input and Its Implications

Now that sound input is supported by the operating system, many more applications can take advantage of what was a specialized capability. What are the implications of sound input? First of all, sound input is another step in the direction of providing operating-system support for multimedia data, sound being one of the basic media forms. Apple is encouraging all developers to support sound in applications if it makes sense.

What can applications (other than the obvious, such as music applications) do with sound? One practical use for sound input is to support an audio note-taking capability, so that users can point to something on a display and dictate a note. Many people reviewing documents might find this a more natural way to give feedback than typing. Another use might be to add a voice mail capability to existing electronic mail systems.

The Advanced Technology Group at Apple is working on a set of human interface guidelines for sound. These guidelines will fit this new sound capability into the existing human interface, without causing any disruptions. Apple will also publish a set of Technical Notes on some of the lower-level facilities, such as editing notes, storing them, representing them on the screen, and compressing them.

▶ Sound Input Devices

The Sound Manager provides a way for sound input devices to register themselves, whether the device is internal (built-in) or external to the Macintosh. If any sound input devices are registered with the Sound Manager, then when the user brings up the Sound control panel, the user will be presented with a scrollable list of these devices. The user can then select the *current sound input device* using the same process as choosing a printer.

The Sound control panel also includes two buttons, Add and Remove, if the application uses any sound input devices. These buttons allow the user to record a new system sound or remove existing sounds. Figure 12-4 shows the Sound control panel. The list of microphones and the "Add..." button are only displayed on Macintoshes that have an installed sound input device.

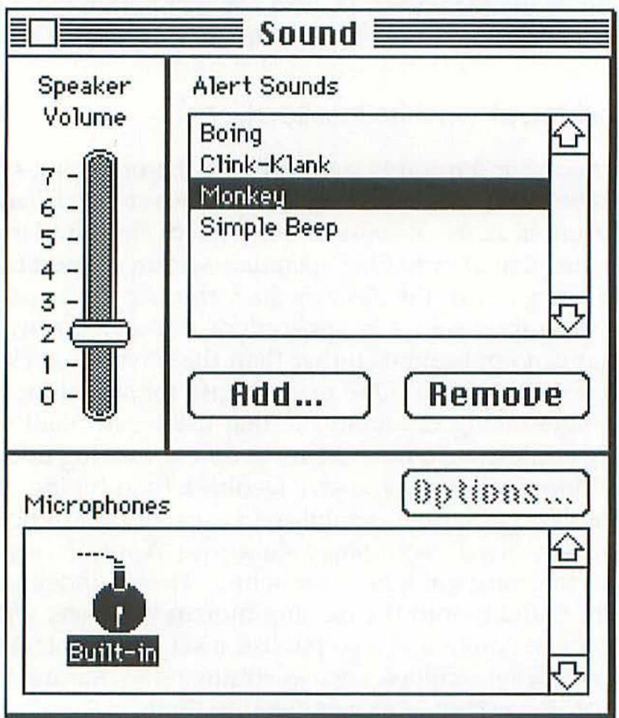


Figure 12-4. The Sound control panel

▶ Using the High-Level Interface to Record Sounds

The high-level interface to record sounds is simple and easy to use—it has only three routines. Call **SndRecord** to record sounds from the currently selected sound input device to memory. Among the parameters for this routine, you'll need to specify where the sound should be stored in memory and the quality of the sound. The three levels of quality are as follows:

- Best—Uses the most memory, but provides the best sound because no compression is done
- Better—Uses less memory by compressing the sound at a 3:1 ratio; suitable for music
- Good—Uses the least amount of memory by compressing the sound at a 6:1 ratio; suitable for voice

SndRecord displays a dialog box that resembles the control panel of a tape recorder. The user can record, pause, stop, and play a sound. When the user is satisfied with the sound, he or she can press the Save button or else use the Cancel button. Figure 12-5 shows the SndRecord dialog box. You can use the **SndPlay** routine, discussed earlier in this chapter, to play back the sound.

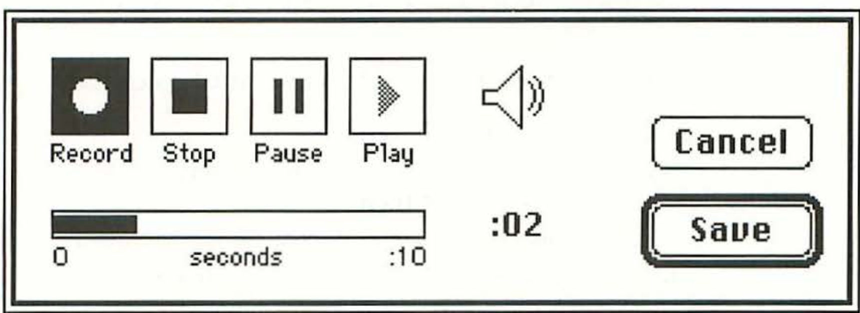


Figure 12-5. The SndRecord dialog box

Call **SndRecordToFile** to record sounds from the currently selected sound input device to a disk file. This routine takes the same parameters as **SndRecord**, except that instead of a handle, you'll give a working directory refNum and a file name. It displays the same dialog as **Snd-Record**.

▶ Using the Low-Level Interface to Record Sounds

The low-level sound input interface is provided for applications that require more control over the sound input process and for sound input device drivers. The low-level routines use the SPB (Sound-input Parameter Block) data structure, which includes fields such as the number of bytes to record (when recording is completed, it contains the number of bytes recorded), the number of milliseconds to record, a pointer to the buffer where recorded sound data should be stored, and a pointer to a completion routine (which is executed when recording terminates).

Call **SPBGetIndexedDevice** to get the name and icon of the sound input device by index. By starting with an index of 1 and continuing until you get an error code of badSoundInDevice, you can get a list of all available sound input devices.

To open a sound input device, call **SPBOpenDevice**. Specify the device by name and, if this call is successful, it will return a sound input reference number. You'll use this number when filling out an SPB record to use with other calls.

Call **SPBGetDeviceInfo** to get information about a sound input device. You must pass a selector (similar to calling **Gestalt**) to get the particular information you're interested in. Call **SPBSetDeviceInfo** to set the state of a sound input device. You'll also need to use a selector when using this call. Examples of selectors are listed in Table 12-3. Some selectors can be used with both calls; others can be used with only one of these calls.

Table 12-3. Examples of sound input selectors

Selector	*Returns*
'SSav'	Number of sample sizes supported by the sound input device and a list of them
'SSiz'	Size of the sample produced by the device
'Icon'	Sound input device icon and mask
'ChAv'	Number of channels available
'Chan'	Number of channels to record
'AGC '	State of Automatic Gain Control

Call **SPBRecord** to start recording sounds to memory, passing an SPB data structure. Call **SPBRecordToFile** to record sounds to a disk file. Three routines allow you to control the recording process: **SPBPause-Recording**, **SPBResumeRecording**, and **SPBStopRecording**.

Two routines are provided that simplify recording sounds into an 'snd ' resource or an AIFF file. Call **SetupSndHeader** or **SetupAIFFHeader**,

passing the many parameters required to create an 'snd ' or AIFF header. The easiest way to use either of these calls is to call them first with a buffer length of 0. In this case, the call will simply calculate the length of the header but not create the header. Reserve this much space in the resource or file, and then record the sound. When the sound has been recorded, make another call to **SetupSndHeader** or **SetupAIFFHeader**, passing the final length of the sound. This time the header will be created (in the appropriate place).

To get the current status of the recording process, call **SPBGetRecordingStatus**. This routine returns parameters such as the number of bytes recorded so far, the number of milliseconds recorded so far, and the meter level (sound level). Two utility routines are provided to convert between time and bytes: **SPBMilliSecondsToBytes** and **SPBBytesToMilliSeconds**.

Three routines are provided for sound input drivers. Call **SPBCloseDevice** to close a sound input device that you previously opened. You need to pass it the sound input reference number of the device you want to close. **SPBSignInDevice** registers a sound input device with the Sound Manager. **SPBSignOutDevice** unregisters a sound input device with the Sound Manager. **SPBGetIndexedDevice** returns the name and icon of the *n*th device.

▶ Conclusion

The Sound Manager offers a wide variety of ways to use sound in applications. Applications that simply play sounds can use the high-level interface to the Sound Manager. Music and sound applications can use the low-level interface for more control over the processing of sound.

The Sound Manager in System 7 provides another new capability, that of recording sounds. An easy-to-use high-level interface is provided for applications, and a low-level interface is provided primarily for sound input drivers.

Get Info ▶

For more information on the Sound Manager, refer to the Sound Manager chapter of *Inside Macintosh*, Volume VI.

The Audio Interchange File Format is available from APDA as a document titled *Audio Interchange File Format*.

The MACE toolkit is also available from APDA for anyone interested in using these routines under System 6. (This toolkit has been incorporated into the Sound Manager under System 7.) This product is called the *Macintosh Audio and Compression Toolkit*.

13 ▶ The Communications Toolbox

▶ Introduction

In this chapter, you will look at the set of managers known collectively as the Communications Toolbox. This toolbox provides a standardized set of communications services that make it easy for you to add file transfer services, terminal emulation, and data connection services to any application.

Applications programmers have often avoided dealing with communications because it seems like a strange and unusual world compared to writing applications software. The Communications Toolbox provides powerful communications functions that you can use without having to do a lot of work. It provides facilities for arbitrating between applications that want to use a communications device.

In this chapter, you'll first look at the architecture of the Communications Toolbox. You'll learn about the various components of the toolbox and how they work together. Then you'll look at each of the major aspects of the toolbox in turn: programming with the Connection Manager, writing a connection tool, programming with the Terminal Manager, writing a terminal (emulation) tool, programming with the File Transfer Manager, writing a file transfer tool, programming with the Communications Resource Manager, and programming with the Communications Toolbox Utilities.

▶ The Architecture of the Communications Toolbox

The Communications Toolbox provides three basic capabilities: connecting with another computer, emulating a terminal, and transferring files using a protocol. The toolbox doesn't actually provide these capabilities,

but rather provides a framework and a programmatic interface for applications to use these capabilities. The Communications Toolbox in System 7 consists of five managers:

- Connection Manager
- Terminal Manager
- File Transfer Manager
- Communications Resource Manager
- Communications Toolbox Utilities

The work of running a connection, emulating a terminal, and transferring a file is not implemented in this toolbox, but in tools that are called by the first three managers. Connection tools, terminal tools, and file transfer tools are separate pieces of code, each of which lives in its own file.

The overall architecture of the Communications Toolbox is illustrated in Figure 13-1. Pieces of code that exist in their own files are drawn with a thicker line. The Communications Toolbox is part of the operating system, starting with System 7. However, it still has the appearance of a separable subsystem.

Notice that an application does not call a tool directly. It always calls one of the managers of the Communications Toolbox and lets the manager call the current tool. Although this is not illustrated here, an application can use more than one tool at a time. That is, an application could use more than one connection tool at a time, or it could use the same tool more than once. At the same time, it can also be using file transfer tools and terminal tools. Other applications can also be using the same tools as your application.

The application and tools can use the routines of the Communications Resource Manager and the Communications Toolbox Utilities. All the various pieces of code use the other routines of the Macintosh operating system and Toolbox (which is different from the Communications Toolbox). The operating system talks to the hardware primarily through device drivers.

Each tool has the same relationship with its manager that a printer driver has with the Print Manager. The majority of the work happens in the tool or driver, but the manager provides a standard interface to the capability.

You can use any of the three capability managers (Connection, Terminal or File Transfer) independently of the other two. For example, you can use the Connection Manager without using either the Terminal Manager or the File Transfer Manager. If you use any of these three managers, however, you end up using the Communications Resource Manager and

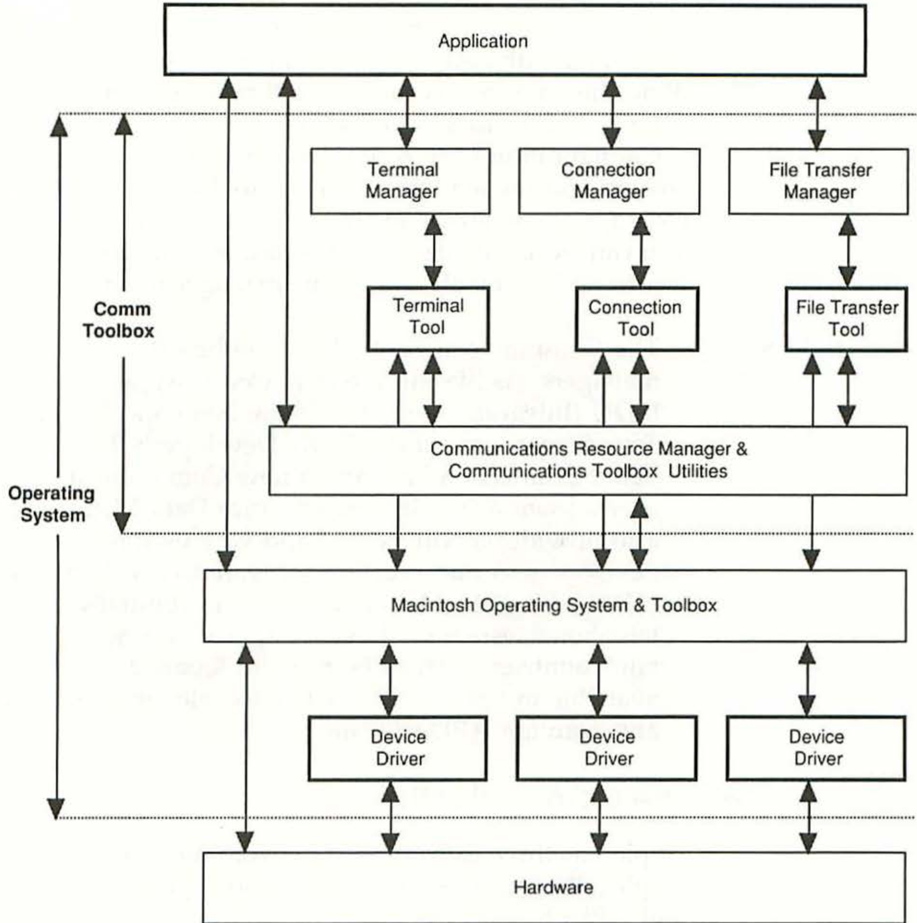


Figure 13-1. The architecture of the Communications Toolbox

the Communications Toolbox Utilities, because the tools almost always use some of the routines in the latter two managers. You must therefore initialize these two managers before initializing and using any of the three capability managers.

▶ Data Structures

Two kinds of data structures are used by the managers of the Communications Toolbox. Connection, terminal, and file transfer records are the first kind of important data structure. Almost every call to their managers

requires a handle to one of these records. These records store all the data associated with each use of one of those managers. This method of handling the data means that those three managers can be used more than once by one or more applications.

Configuration records, the second important type of data structure, are created and owned by tools. Each tool has its own unique configuration record; the contents and size of the record may differ from tool to tool. You can store a configuration record in a document file. Before using the record again, you should ask its manager to validate it.

Note ▶ The Communications Toolbox will be extended to encompass other managers. As System 7 was released, Apple was developing an ISDN (Integrated Services Digital Network) Developer's Toolkit. Two components of the ISDN Developer's Toolkit are an ISDN Serial Connection Tool and a new Communications Toolbox manager known as the Integrated Voice Data Manager. This toolkit also provides a NuBus card and various software tools to enable developers to start creating software that will take advantage of ISDN, a digital technology that will eventually replace the analog telephone system we have today. ISDN is widely available in certain countries, such as France and Japan, but is slowly becoming available in the U.S. The ISDN Developer's Toolkit will be available through APDA in the future.

▶ The Communications Toolbox in the Larger Picture

Apple has three categories of networking and communications software: AppleTalk, packages to support various protocols (such as X.25, TCP/IP, and APPC), and now the Communications Toolbox. There is little connection between the software in any of these categories. If you are an application programmer, then you'll have to learn the details of each of these packages separately; the application programming interface (API) for every one of these packages is completely different from all the others.

▶ Compatibility and the Communications Toolbox

Call **Gestalt** with a selector of gestaltCTBVersion to find out the current version of the Communications Toolbox. Version 2 of the toolbox was shipped as part of System 7.

Before using any of the Communications Toolbox managers, verify that these managers are available on the current Macintosh. Do this by calling **Gestalt** with the following selectors:

- gestaltConnMgrAttr for the attributes of the Connection Manager
- gestaltCRMAttr for the attributes of the Communication Resource Manager
- gestaltFXfrMgrAttr for the attributes of the File Transfer Manager
- gestaltTermMgrAttr for the attributes of the Terminal Manager

Under System 7, the only attribute returned for these four selectors is that the appropriate manager is available or not available.

Each of the managers in the Communications Toolbox has a routine for getting its version number. You have to check that each of these calls exists before using them, so calling **Gestalt** is less work. These routines are as follows:

- **CMGetCMVersion** (Connection Manager)
- **TMGetTMVersion** (Terminal Manager)
- **FTGetFTVersion** (File Transfer Manager)
- **CRMGetCRMVersion** (Communications Resource Manager)
- **CTBGetCTBVersion** (Communications Toolbox Utilities)

Version 2 of each of these managers was shipped as part of System 7. If you're using **Gestalt** to get the version number of the Communications Toolbox, then you don't need to use these calls.

▶ Programming with the Connection Manager

The Connection Manager provides connection services between one process and another. The other process can be on another machine (it can be another Macintosh or something else) or on the same machine. The services provided by the Connection Manager are independent of protocols. Protocols are handled by a lower level of code known as connection tools, which are described in the next section. The Connection Manager does not provide support for transmission error detection and correction or for flow control, which can all be provided by the connection tool. An example of a connection tool is shown in Figure 13-2.

The Connection Manager supports up to three channels per connection: the data, attention, and control channels. The data channel is the

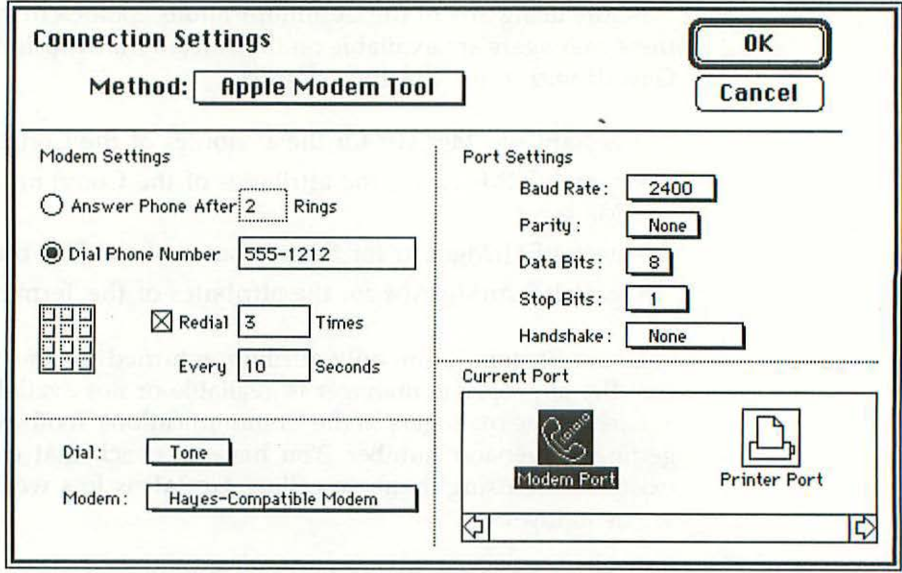


Figure 13-2. Set up Dialog presented by the Apple Modem connection tool

primary channel for communicating over the connection. Depending on the connection tool, the other two channels may or may not be implemented. You can use the attention channel to warn the other end of the connection that you're closing the connection. The control channel can be used to keep the connection control information separate from the data flowing over the connection.

Connection tools bear a similar relationship to the Connection Manager that printer drivers have with the Print Manager. Figure 13-3 illustrates this analogy. The Terminal and File Transfer Managers also fit into this analogy. An application calls the Print Manager, which performs relatively little work. The Print Manager calls the printer driver selected by the user to do the work of printing. In a similar way, an application calls the Connection Manager, which performs relatively little work. The Connection Manager calls the connection tool selected by the user to operate the connection.

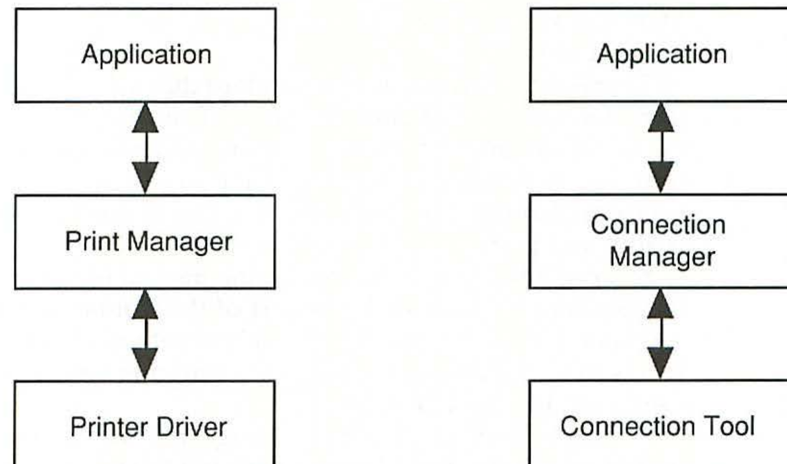


Figure 13-3. The relationship between connection tools and the Connection Manager

▶ The Connection Record

Applications pass data to the routines of the Connection Manager using a *connection record*. This protocol-independent record contains all the details required to manage a connection, such as which channel to use (data, attention, and/or control), a reference constant, a long word for user data, and pointers to the I/O buffers for the channels used. This record also contains configuration records for the selected connection tool to store its private data. The connection record is private to the Connection Manager; you are only allowed to directly modify the configuration record. The Connection Manager provides a set of routines for modifying some of the contents of the record.

An application can use more than one connection at a time. Your application will need to allocate a connection record for each connection that your application uses.

▶ Using the Connection Manager Routines

Call **InitCM** to initialize the Connection Manager. You should have already called **InitCRM** to initialize the Communications Resource Manager and **InitCTBUtilities** to initialize the Communications Toolbox Utilities. You must initialize these two managers before using the File Transfer

Manager, whether you will use any of the routines in the latter two managers or not.

To present the user with the standard dialog box for choosing a connection tool, call the **CMChoose** routine. This dialog will search for all connection tools in the Communications Folder in the System Folder. The user selects which tool to use from a pop-up menu on this dialog. The user can also configure the tool from this dialog box, such as baud rate, parity, and port.

You can also create a custom dialog instead of using the standard dialog. Six routines provided as a part of the Connection Manager simplify this code to a certain extent. One other method of selecting a connection tool is to have your application use a scripting language. In this case, the user wouldn't see a dialog box.

To interface with a scripting language, use the two routines provided as part of the Connection Manager that simplify this process: **CMGetConfig** and **CMSetConfig**. These routines get a configuration string from and set the configuration string for a connection tool, respectively. Remember that these calls are being processed by the connection tool; the Connection Manager does nothing more than provide an API to the tool with these calls.

▶ Preparing to Use a Connection

Use the **CMNew** routine to create a new connection record. When making this call, pass the address of the four procedures in your code that will send data, receive data, read data from a file, and write data to a file. The routines that send and receive data can use the Connection Manager to perform their function. You'll also need to pass the pointer to a procedure that the connection tool can call to find out what environment it's running in, such as how wide the data channel is in bits. Last, you'll need to pass the procID of the connection tool. Get this value by calling **CMGetProcID**. You'll need to specify the name of the connection tool to get its procID, which is assigned dynamically by the Connection Manager.

As part of processing the **CMNew** call, the Connection Manager calls **CMDefault** to fill in the configuration records in the connection record. The Connection Manager also loads the main procedure of the connection tool (see the next section for details on connection tools).

Use the **CMDefault** routine to ask the current connection tool to (allocate and) set the connection record to default values. Use the **CMValidate** routine to verify that the connection record is internally consistent.

▶ Opening, Managing, and Closing Connections

Call **CMOpen** to open a connection. The Connection Manager uses the information in the connection record to accomplish this. You can specify whether the open request should be made synchronously or asynchronously. If, on the other hand, you want to wait for incoming requests to open a connection, call **CMListen**. This call can also be made synchronously or asynchronously, but the latter case is the normal way to use this call. You can tell when an incoming request has been made by using the **CMStatus** call. If there is an incoming request, then the cmStatusIncomingCallRequest bit will be set. To accept or reject an incoming request, call **CMAccept**. Note that you cannot call this routine at interrupt level.

Your application should call **CMIdle** every time it passes through the main event loop. This routine will give the tool handling the specified connection some time to perform its idle-loop tasks.

You can use the **CMStatus** call to find out the current status of a connection. In addition to telling you if there is an outstanding request for a connection, this routine also tells you whether the following conditions occur:

- The connection is in the process of being opened, open, or being closed
- Data is present on the data, attention, and/or control channels
- A read and/or a write is pending on the data, attention, and/or control channels
- The tool is breaking the connection

To get information on the connection environment, use the **CMGetConnEnvirons** routine. This call returns the following information.

- The data transfer rate
- The width of the channel in bits
- Which channels are supported by this connection (data, attention, and/or control)
- Whether hardware or software flow control is in use

This call is useful when using the Terminal or File Transfer Manager. Your application may want to use them differently, depending on whether you can use the three channels or only one, or depending on the data transfer rate.

Call **CMAbort** if you need to cancel a pending open operation (having used **CMOpen** asynchronously) or to cancel listening for an incoming connection request (having used **CMListen** asynchronously).

You can call **CMClose** to close a connection whether the connection is in the process of being opened or has been opened. This call does not deallocate memory; call **CMDispose**, when you're completely finished with the connection, to deallocate the connection record and the data structures to which it refers.

▶ Reading, Writing, and Searching over a Connection

The **CMRead** and **CMWrite** routines perform the basic work of reading and writing over a connection. Actually, these routines read from and write to buffers; the connection tool performs the work of sending or receiving this data over the connection. With both of these routines, you supply a connection record, the channel, an I/O buffer, a timeout value, the address of a completion routine (which will be called when the request has been completed), a flag indicating whether to send an end-of-message signal or whether you've received one, and a flag indicating whether this routine should be processed synchronously or asynchronously.

You can have a simultaneous read and write request on a channel of a connection. You can't have more than one read request at a time on a given channel of a connection. This also holds true for write requests.

Use the **CMIOKill** routine to stop a pending read or write request on the specified channel (data, attention, or control).

Call **CMBreak** to cause a "break" operation. This operation makes sense with some protocols, but not for others. Call **CMReset** to reset the connection. The meaning of this operation also depends on the protocol.

You can ask the Connection Manager to search for a string of up to 255 bytes in the incoming stream by calling **CMAddSearch**. The Connection Manager adds this search to its list, and calls your callback routine when it finds the specified string. Use the **CMRemoveSearch** routine to remove a specific search request, or the **CMClearSearch** routine to remove all outstanding search requests on a particular connection.

▶ Handling Events with the Connection Manager Routines

Use the **CMEvent** routine to tell the connection tool that the event your application just received happened in a window owned by the tool. The tool should therefore process the event.

To tell the connection tool that the menu event your application just received happened in a menu owned by the tool, use the **CMMenu** routine. The tool should therefore process the event.

You use the **CMActivate** routine to tell the connection tool that your application has received an activate or deactivate event. The tool may

need to do something when this happens, such as installing or removing a menu.

Use the **CMResume** routine to tell the connection tool that your application has received a suspend or resume event. When your application is going to be running in the background (or returning to the foreground), the tool may adjust some of its parameters.

▶ Other Connection Manager Routines

Call **CMGetToolName** to get the name of the specified connection tool. If you want to save the connection tool's configuration, you also need to save the name of the tool.

Use the **CMGetRefCon** and **CMSetRefCon** routines to get and set the reference constant in the connection record. Use the **CMGetUserData** and **CMSetUserData** routines to get and set the user data (it's 4 bytes, so it can be a handle) in the connection record.

Call **CMGetVersion** to get the 'vers' information from the connection tool. This information is returned in a handle, which you must dispose of when you're done with it.

▶ Internationalization and the Connection Manager

To help internationalize your application, use **CMIntlToEnglish** and **CMEnglishToIntl**, which ask the file transfer tool to translate configuration strings from and to other languages. These routines assume that the strings are stored in the tool in American English.

▶ Writing a Connection Tool

A connection tool implements a particular connection protocol. Connection tools have a structure that makes them interchangeable from the user's (and application's) point-of-view. That is, it it easy for the user to switch from using one connection tool to another. Connection tools bear a similar relationship to the Connection Manager that printer drivers have with the Print Manager: An application calls the appropriate manager, which usually performs relatively little work and calls the tool or driver to accomplish most of the work for this call.

The relationship between the application and various components of the Communications Toolbox is illustrated in Figure 13-4. An application might call the Connection Manager, which would then call the current connection tool. The connection tool might then talk with a device driver controlling the communications hardware. This figure also illustrates a

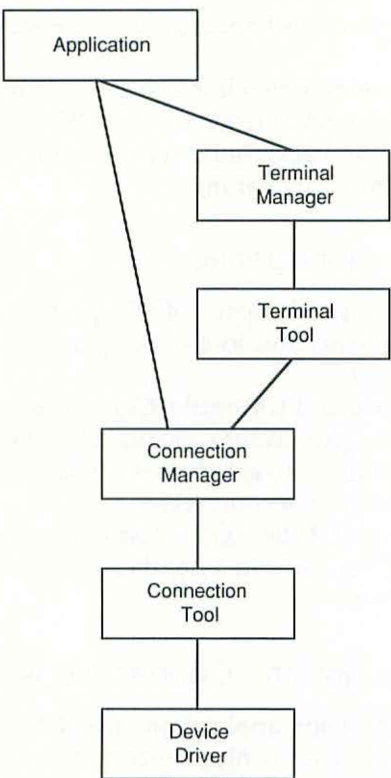


Figure 13-4. The relationship between an application and components of the Communications Toolbox

more complex example. An application can call the Terminal Manager, which could call a terminal tool. This tool might call the Connection Manager, which then calls a connection tool, which then calls the device driver.

Connection tools are structured as a set of resources in a single file. A connection tool has a basic set of five or six resources.

- 'cdef' code resource (not the same as a 'CDEF' resource!)—Implements the communications functions of the tool.
- 'cval' code resource—Validates connection records when the Connection Manager is called using the **CMValidate** call. This code also handles the **CMDefault** call.
- 'cset' code resource—Handles the tool-settings dialog.

- 'cscr' code resource—Handles the interface between a scripting language and the tool.
- 'cloc' code resource—Translates this tool's strings between English and other languages.
- 'cbnd' bundle resource (optional)—Contains the tool's name and lists all the resources in this tool.

All of the code resources ('cdef', 'cval', 'cset', 'cscr', and 'cloc') are called by the Connection Manager. The Connection Manager passes along a message, which is a code telling the code resource what to do, and a set of three parameters. The meaning of the parameters is determined by the message. Not all parameters are used for all messages.

These are not the only resources that are in a connection tool. The other resources needed include the resources for dialog boxes and dialog item lists. You'll now look at creating each of the basic six resources in turn.

▶ Creating a 'cdef' Resource

The 'cdef' resource performs the primary functions of the connection tool: setting up, using, and closing down a connection using a particular protocol.

A connection tool might be called by the Connection Manager with any of the messages listed in Table 13-1. If your tool does not understand the message or cannot support it, then it should return an error message, cmNotSupported. Note that the Connection Manager's caller can be either an application or another communications tool.

Let's now look at several basic messages that every connection tool should support: cmInitMsg and cmDisposeMsg, cmOpenMsg and cmCloseMsg, and cmReadMsg and cmWriteMsg.

The cmInitMsg message tells your code to initialize for a new connection. This is the time to tell the caller whether this tool supports a data channel, an attention channel, and/or a control channel. You may need to allocate an input and an output buffer. The cmDisposeMsg message tells your code to deallocate a local memory (which was allocated when you received the cmInitMsg message). The Connection Manager handles deallocating the configuration records and the connection record. If you try to deallocate them, you'll cause a system crash.

The cmOpenMsg message tells your tool to open a connection. You'll be passed a timeout value (in clock ticks) and perhaps a CMCompletorRecord record, which specifies whether the operation should be performed synchronously or asynchronously. This record contains a pointer to a completion routine that you must call when the open operation has been completed asynchronously.

Table 13-1. Messages for the 'cdef' resource of a connection tool

Message Name	*Sent to the Connection Tool When the Caller:*
cmInitMsg	Initializes the connection
cmDisposeMsg	Is closing the connection
cmSuspendMsg	Requires the tool to handle a suspend event
cmResumeMsg	Requires the tool to handle a resume event
cmMenuMsg	Got a menu event for a menu belonging to this tool
cmEventMsg	Got a window event and this event is associated with this tool
cmActivateMsg	Requires the tool to handle an activate event
cmDeactivateMsg	Requires the tool to handle a deactivate event
cmIdleMsg	Has idle time (but never at interrupt time)
cmResetMsg	Requires your tool to reset the connection
cmAbortMsg	Requires that a pending open or listen be aborted
cmReadMsg	Reads from the connection
cmWriteMsg	Writes to the connection
cmStatusMsg	Wants to know the status of the connection from the tool (such as whether a read is pending or a write is pending)
cmListenMsg	Requires the tool to listen for an incoming connection request
cmAcceptMsg	Has accepted the connection
cmCloseMsg	Closes a connection
cmOpenMsg	Opens a connection
cmBreakMsg	Sends a break message
cmIOKillMsg	Wants to kill any outstanding read or write requests
cmEnvironsMsg	Wants information on the connection (such as baud rate and the number of data bits)

The cmCloseMsg message tells your tool to close a connection. You'll be passed a timeout value (in clock ticks) and perhaps a CMCompletorRecord record. You'll need to close the channel and perhaps close the input and output drivers.

The cmReadMsg or cmWriteMsg message tells your connection tool to read from or write to the connection. In either case, you'll be passed a CmDataBuffer record, a CMCompletorRecord record, and a timeout value. The CMDataBuffer record contains the address of the buffer to read from (or write to), the number of bytes to read or write, which channel to use (data, attention, or control), and an end-of-message flag. If the timeout value is 0, you should complete as much of the operation as possible and return.

▶ Creating a 'cval' Resource

The 'cval' code resource validates connection records. It must handle two messages from the Connection Manager: cmValidateMsg and cmDefaultMsg. If it receives a message it cannot understand, the code should return a message of cmNotSupported.

When processing a cmValidateMsg message, check the connection record. If the record is valid, return 0; otherwise, rebuild the configuration record and return 1.

When processing a cmDefaultMsg message, set the connection record to the default values. Also, if requested, you should allocate a new configuration record.

▶ Creating a 'cset' Resource

The 'cset' code resource handles a dialog box that allows a user to select and configure a connection. This code must handle five messages: cmSpreflightMsg, cmSsetupMsg, cmSitemMsg, cmSfilterMsg, and cmScleanupMsg. If it receives a message it cannot understand, the code should return a message of cmNotSupported.

The cmSpreflightMsg message tells your setup code to initialize whatever local variables are needed for the configuration process and to get any required resources.

Use the cmSsetupMsg message to have your setup code initialize the configuration dialog box to the current configuration. The cmSitemMsg message tells your setup code when an item was selected by the user from the configuration dialog. Process the message for that dialog item.

You use the cmSfilterMsg message to have your setup code filter events. Your code can handle the event (in which case you should return 1) or ignore it (in which case you should return 0). The cmScleanupMsg message tells your setup code to clean up by deallocating any temporary memory required for the configuration process.

▶ Creating a 'cscr' Resource

The 'cscr' code resource handles the interface between the connection tool and a scripting language. It must handle two messages: cmMgetMsg and cmMsetMsg. If it receives a message it cannot understand, the code should return a message of cmNotSupported. Note that no standard scripting language exists at this time, but you can use various proprietary scripting languages—communications programs (such as Software Venture's Microphone II) especially seem to use them.

The cmMgetMsg message tells your scripting language interface code to

return a string description of the current connection record in American English. If this description is required in another language, then the string will be passed for translation to the 'cloc' resource, which will be described shortly.

The cmMsetMsg message tells your setup code to parse a string and set the configuration record accordingly. The string passed to your code will be in American English. It might have been translated previously using the 'cloc' resource. Your code should return a 0 if there was no problem doing this, a −1 if there was a generic error, a number less than −1 if there was an operating-system error, and a positive number if you couldn't parse the entire string (the number being the offset to the last character that you successfully parsed). The Connection Manager automatically calls **CMValidate** if this message was successfully processed.

▶ Creating a 'cloc' Resource

The 'cloc' code resource translates messages for your connection tool between American English and other languages. It must handle two messages: cmL2English and cmL2Intl. If it receives a message it cannot understand, the code should return a message of cmNotSupported.

The cmL2English message tells your localization code to return an American English translation of a string in the specified language. The cmL2Intl message tells your localization code to return a translation of a string into the specified language from American English.

▶ Creating a 'cbnd' Resource

The 'cbnd' optional bundle resource contains the tool's name and lists all the resources in this tool. The name of the tool is the name of this bundle resource.

This resource also contains a list of all the other resources in this file, sorted by resource type and containing a local resource ID and an actual resource ID. The code in the connection tool can use the local IDs when referring to these resources. To convert between these IDs and the actual IDs, the Communications Resource Manager uses this 'cbnd' resource, obviously simplifying the code for connection tools.

▶ Programming with the Terminal Manager

The Terminal Manager provides a standard API to terminal-emulation services that is independent of which terminal is being emulated. The details of emulating a particular terminal are handled by a lower level of code known as terminal tools, which are described in the next section. Since the interface provided by the Terminal Manager is independent of

the terminal being emulated, a user should be able to easily switch from emulating one terminal to another. Your application should work as well with one terminal tool as another. Figures 13-5, 13-6, 13-7, and 13-8 show the dialogs presented by the VT102 terminal emulation tool. This tool has four dialogs because of the complexity of the VT102 terminal.

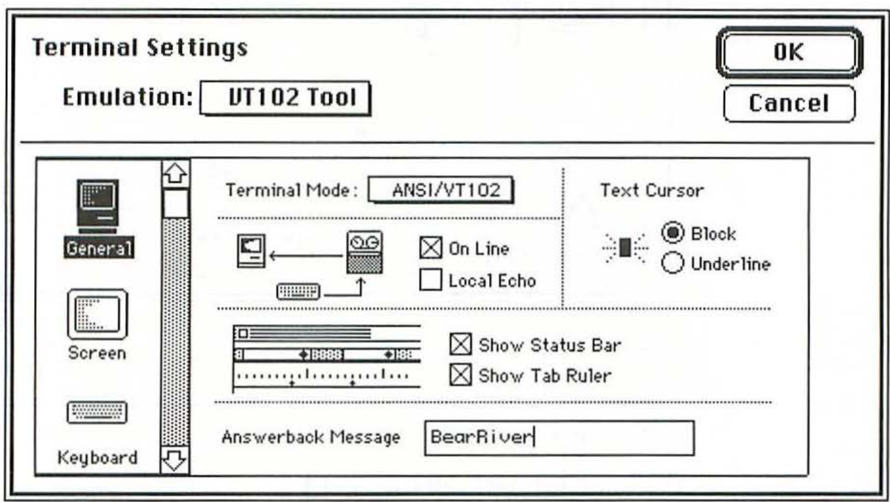


Figure 13-5. General characteristics dialog

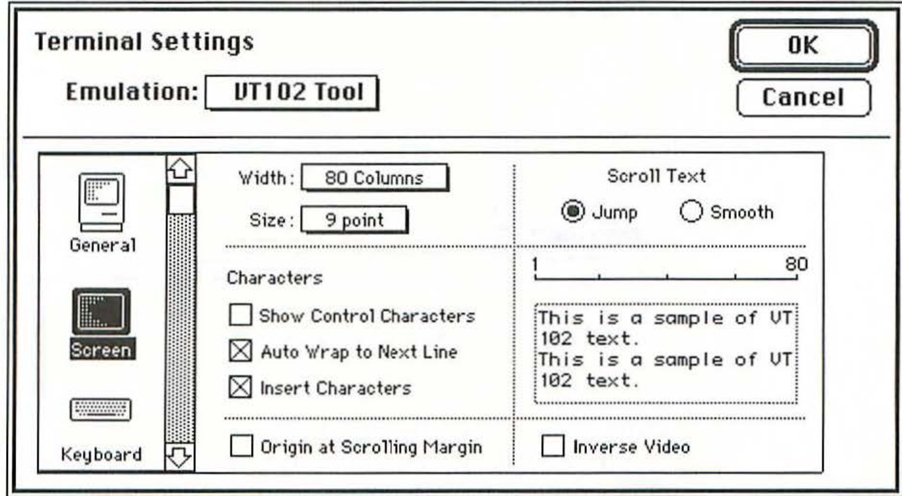


Figure 13-6. Screen parameters dialog

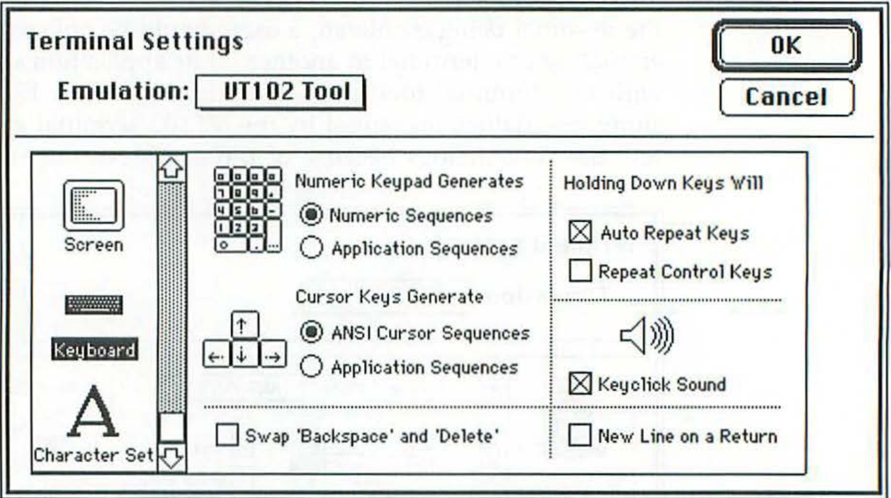


Figure 13-7. Keyboard dialog

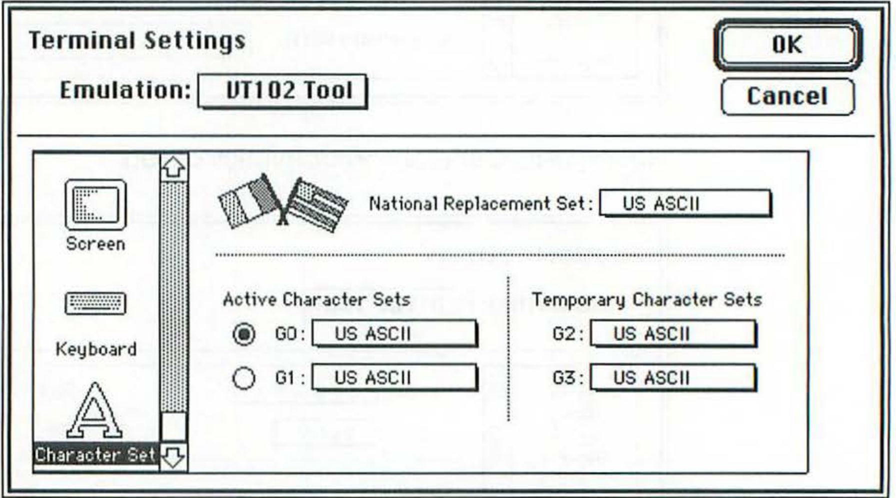


Figure 13-8. Character set dialog

These dialogs handle the emulation's general characteristics, screen parameters, keyboard, and character set.

Terminal tools bear a similar relationship to the Terminal Manager that printer drivers have with the Print Manager; Figure 13-3 illustrates this analogy using the Connection Manager. An application calls the Print

Manager, which performs relatively little work. The Print Manager calls the printer driver selected by the user to do the work of printing. In a similar way, an application calls the Terminal Manager, which performs relatively little work. The Terminal Manager calls the terminal tool selected by the user to do the work of emulating a terminal.

The Terminal Manager provides a window for emulating a terminal. The content area of the window is divided into two parts: the *terminal emulation region* and the *cache region*. The terminal emulation region is where data is displayed emulating a particular terminal. Both text and graphics terminals can be emulated. Your application and the terminal tool use a **TermDataBlock** call to communicate the contents of the terminal emulation region to each other. For text terminals, a **TermDataBlock** call describes a line of text; for graphics terminals, a **TermDataBlock** call describes a picture.

The cache region of a terminal emulation window provides an area you can use to display data that has scrolled off the top of the terminal emulation region. The cache region is optional. If you want it, though, you have to write the code.

You can emulate more than one terminal in an application. You can use the same or more than one terminal tool to accomplish this. Also, because of the architecture of the Communications Toolbox, more than one application can use the same terminal tool at the same time.

▶ The Terminal Record

Applications call the routines of the Terminal Manager passing a *terminal record*. This record, which is independent of the emulation, contains all the details required to support the emulation, such as a reference constant, a long word for user data, the boundaries of the terminal emulation area in its window, and the current selection in the window. Also, the terminal record contains a pointer to the five procedures that can describe the connection environment, handle mouse clicks, transmit data, perform a break operation, and collect lines that scroll off the top of the terminal emulation window. In addition, this record provides pointers to data belonging to the terminal tool: configuration records and an area to store its private data.

The terminal record is private to the Terminal Manager; you can directly modify only the configuration record. The Terminal Manager provides a set of routines for modifying some of the contents of the record.

An application can use more than one terminal emulation at a time. Your application will need to allocate a terminal record for each terminal emulation that your application uses.

▶ Using the Terminal Manager Routines

Call **InitTM** to initialize the Terminal Manager. You should have already called **InitCRM** to initialize the Communications Resource Manager, and **InitCTBUtilities** to initialize the Communications Toolbox Utilities. You must initialize these two managers before using the Terminal Manager, whether you will use any of the routines in the latter two managers or not.

To present the user with the standard dialog box for choosing a terminal tool, call the **TMChoose** routine. This dialog searches for all terminal tools in the Communications Folder in the System Folder, and the user selects which tool to use from a pop-up menu on this dialog. The user can also configure the tool from this dialog box, such as what the cursor looks like, whether the terminal in online or offline, and so on, depending on the terminal tool.

You can also create a custom dialog instead of using the standard dialog. Six routines provided as a part of the Terminal Manager simplify this code to a certain extent. There is one other method of selecting a tool: your application can also select a terminal tool by using a scripting language. In this case, the user wouldn't see a dialog box.

To interface with a scripting language, use the two routines provided as part of the Terminal Manager that simplify this process: **TMGetConfig** and **TMSetConfig**. These routines get a configuration string from and set the configuration string for a terminal tool. Remember that these calls are being processed by the terminal tool; the Terminal Manager is doing nothing more than providing an API to the tool with these calls.

▶ Preparing to Emulate a Terminal

Use the **TMNew** routine to create a new terminal record. When making this call, pass the address of the procedures in your code that will send out data, collect data scrolling off the top of the terminal emulation window, perform a break operation, and handle mouse clicks. You'll also need to pass a pointer to a procedure that the terminal tool can call to find out what connection environment it's running in, such as how wide the data channel is in bits. Last, you'll need to pass the procID of the terminal tool. Get this value by calling **TMGetProcID**. You need to specify the name of the terminal tool to get its procID; these IDs are assigned dynamically by the Terminal Manager.

As part of processing the **TMNew** call, the Terminal Manager calls **TMDefault** to fill in the configuration records in the terminal record. The Terminal Manager also loads the main procedure of the terminal tool (see the next section for details on terminal tools).

Call the **TMDefault** routine to ask the current terminal tool to (allocate and) set the terminal record to default values. Use the **TMValidate** routine to verify that the terminal record is internally consistent.

▶ Emulating a Terminal

Your application should call **TMIdle** every time it passes through the main event loop. This routine will give the tool handling the specified terminal tool some time to perform its idle-loop tasks. Terminal tools cause the cursor to blink and perform searches during this time.

Use the **TMStream** routine to pass the terminal tool data that have come in from the other end of the connection. Call the **TMPaint** routine to draw data in the specified TermDataBlock data structure into the specified rectangle in a terminal emulation window.

To get a line of data from the terminal emulation buffer, use the **TMGetLine** routine. You pass a TermDataBlock data structure that the terminal tool fills in. You must allocate this data structure and deallocate when you're done with it.

Call **TMScroll** to scroll the terminal emulation region horizontally and/or vertically. Use **TMResize** to resize the terminal emulation region to the specified coordinates. You use **TMClear** to clear the terminal emulation region and put the cursor in its home position. This call does not transmit any characters.

To get the current position of the cursor, call **TMGetCursor**. You can request the cursor position either in terms of rows and columns (for text terminal emulation) or in terms of pixels (for graphics terminal emulation).

Call **TMReset** to reset the terminal emulation window. This causes the terminal tool to reset its state, and the configuration record is reset to the last saved state.

You can ask the Terminal Manager to search for a string of up to 255 bytes in the terminal emulation by calling **TMAddSearch**. You can specify whether the search should be case-sensitive and whether to search for diacritical marks. You can also request the type of selection: for text, you can request a standard text selection or a boxlike selection. The Terminal Manager adds this search to its list, and calls your callback routine when it finds the specified string. Use the **TMRemoveSearch** routine to remove a specific search request or the **TMClearSearch** routine to remove all outstanding search requests.

Use the **TMGetSelect** and **TMSetSelection** routines to get the current selection in the terminal emulation window and set the current selection.

Call **TMDispose** when you're completely finished with the terminal

emulation. This routine deallocates the terminal record and the data structures to which it refers.

▶ Working with Special Keys

Some terminals have keys that perform special functions, such as PF1 and Home. Call **TMCountTermKeys** to find out how many special keys the current terminal tool supports. You can then call **TMGetIndTermKey** to get the name of the special key specified by an index.

Call **TMDoTermKey** when you are emulating a special key, passing the name of the special key. The terminal tool will emulate the sequence of bytes that key produces.

▶ Handling Events with the Terminal Manager Routines

Use the **TMKey** routine to process key-down or autokey events. This routine passes control to the terminal tool, which translates the character into a sequence of one or more bytes to be sent out. This is one of the routines you will use most often.

Call the **TMClick** routine to process a mouse-down event. This routine calls your click-handling routine, the address of which you supplied in the terminal record.

To force a part or all of the terminal emulation region to be updated, use the **TMUpdate** routine.

Use the **TMEvent** routine to tell the terminal tool that the event your application just received happened in a window owned by the tool. The tool should therefore process the event.

Call the **TMMenu** routine to tell the terminal tool that the menu event your application just received happened in a menu owned by the tool. The tool should therefore process the event.

Use the **TMActivate** routine to tell the terminal tool that your application has received an activate or deactivate event. The tool may need to do something when this happens, such as installing or removing a menu.

To tell the terminal tool that your application has received a suspend or resume event, use the **TMResume** routine. When your application is going to be running in the background (or returning to the foreground), the tool may adjust some of its parameters.

▶ Other Terminal Manager Routines

Use the **TMGetTermEnvirons** routine to get information on the terminal emulation environment. This call returns the following information from the terminal tool:

- If the terminal being emulated is a text and/or graphics terminal
- The number of rows and columns in the terminal emulation region
- The height and width of each cell
- The height and width of the graphics terminal emulation area

To get the name of the specified terminal tool, call **TMGetToolName**. If you want to save the terminal tool's configuration, you'll also need to save the name of the tool.

Use the **TMGetRefCon** and **TMSetRefCon** routines to get and set the reference constant in the terminal record. Use the **TMGetUserData** and **TMSetUserData** routines to get and set the user data (it's 4 bytes, so it can be a handle) in the terminal record.

Call **TMGetVersion** to get the 'vers' information from the terminal tool. This information is returned in a handle that you must dispose of when you're done with it.

▶ Internationalization and the Terminal Manager

Use **TMIntlToEnglish** and **TMEnglishToIntl** to help internationalize your application. These routines, which assume that the strings are stored in the tool in American English, will ask the terminal tool to translate configuration strings from and to other languages.

▶ Writing a Terminal (Emulation) Tool

A terminal tool emulates a particular terminal, such as a DEC VT102. Terminal tools have a structure that makes them interchangeable from the user's (and application's) point-of-view. That is, it is easy for the user to switch from using one terminal tool to another. Terminal tools bear a similar relationship to the Terminal Manager that printer drivers have with the Print Manager. An application calls the appropriate manager. This manager usually performs relatively little work and calls the tool or driver to accomplish most of the work for this call.

Terminal tools are structured as a set of resources in a single file. A terminal tool has a basic set of five or six resources.

- 'tdef' code resource—Implements the communications functions of the tool.
- 'tval' code resource—Validates terminal records when the Terminal Manager is called using the **TMValidate** call. This code also handles the **TMDefault** call.

- 'tset' code resource—Handles the terminal-settings dialog.
- 'tscr' code resource—Handles the interface between a scripting language and the tool.
- 'tloc' code resource—Translates this tool's strings between English and other languages.
- 'tbnd' bundle resource (optional)—Contains the tool's name and lists all the resources in this tool.

All of the code resources ('tdef', 'tval', 'tset', 'tscr', and 'tloc') are called by the Terminal Manager. The Terminal Manager passes along a message, which is a code telling the code resource what to do, and a set of three parameters. The meaning of the parameters is determined by the message. Not all parameters are used for all messages.

The 'tval', 'tset', 'tscr', 'tloc', and 'tbnd' resources are identical in function and structure to the 'cval', 'cset', 'cscr', 'cloc', and 'cbnd' resources described previously in this chapter, in the section on "Writing a Connection Tool." The prefix for the names of messages, error codes, and data structures associated with these resources is *tm* rather than *cm*. For the details on creating these resources, refer to the previous section. These are not the only resources in a terminal tool; also needed are the resources for dialog boxes and dialog item lists. You'll now look at creating a 'tdef' resource.

▶ Creating a 'tdef' Resource

A 'tdef' resource performs the primary functions of the terminal tool: emulating a particular type of terminal. A terminal tool might be called by the Terminal Manager with any of the messages listed in Table 13-2. If your tool does not understand the message or cannot support it, then it should return an error message, tmNotSupported. Note that the Terminal Manager's caller can be either an application or another communications tool.

You'll now look at several of the basic messages that every terminal tool should support: tmInitMsg and tmDisposeMsg, tmStreamMsg, tmKeyMsg, tmDoTermKeyMsg, and tmClickMsg.

The tmInitMsg message tells your code to initialize for a new terminal emulation session. This is the time to allocate any local memory for this tool, including buffers. The tmDisposeMsg message tells your code to deallocate any local memory that was allocated when you received the tmInitMsg. The Terminal Manager handles the deallocation of the configuration records and the terminal record. If you try to deallocate them, you'll cause a system crash.

Use the tmStreamMsg message to send your code a buffer of incoming

data that should be processed and added to the terminal emulation buffer. You'll also be passed the buffer length and some flags that tell you about the contents of the buffer.

Table 13-2. Messages for the 'tdef' resource of a terminal tool

Message Name	*Sent to the Terminal Tool When the Caller:*
tmInitMsg	Initializes the terminal emulation
tmDisposeMsg	Closes the terminal emulation
tmSuspendMsg	Requires the tool to handle a suspend event
tmResumeMsg	Requires the tool to handle a resume event
tmMenuMsg	Got a menu event for a menu belonging to this tool
tmEventMsg	Got a window event and this event is associated with this tool
tmActivateMsg	Requires the tool to handle an activate event
tmDeactivateMsg	Requires the tool to handle a deactivate event
tmIdleMsg	Has idle time (time to make the cursor blink)
tmResetMsg	Requires your tool to reset the terminal emulation window
tmKeyMsg	Got a key-down, key-up, or autokey event
tmStreamMsg	Has a buffer of data to be added to the terminal emulation buffer
tmResizeMsg	Needs to resize the terminal emulation window
tmUpdateMsg	Wants the terminal emulation window updated
tmClickMsg	Got a mouse-down event in the terminal emulation window
tmGetSelectionMsg	Wants the selection in the terminal emulation window returned (for a cut, copy, or paste operation)
tmSetSelectionMsg	Tells you what the current selection is (for a cut, copy, or paste operation)
tmScrollMsg	Wants the terminal emulation window to scroll horizontally or vertically
tmClearMsg	Wants the terminal emulation window cleared (and all buffers cleared)
tmGetLineMsg	Wants the data, character attributes, and line attributes for one or more lines of text in the terminal emulation window

Table 13-2. Messages for the 'tdef' resource (continued)

Message Name	*Sent to the Terminal Tool When the Caller:*
tmPaintMsg	Wants to replace one or more lines of text in the terminal emulation window and have them displayed
tmCursorMsg	Wants the location of the cursor
tmGetEnvironsMsg	Wants the current environment of the terminal emulation window
tmDoTermKeyMsg	Pressed a special key (such as PF1 or Home)
tmCountTermKeysMsg	Wants the number of special keys supported by this tool
tmGetIndTermKeysMsg	Wants the name of the specified special key

You use the tmKeyMsg message to pass your code the key-down, key-up, or autokey event that the application received. You aren't passed the event record, but rather the character code, key code, and modifier (such as Shift or Option). Your code should process this keystroke—it might have to remap the character, transmit the character, and echo the character to the screen.

The tmDoTermKeyMsg message passes your code a string containing the name of a special key (such as PF1 or Home) that was pressed. Your code needs to perform the functions associated with that key. Note that the application previously got the name of this key from your tool using the tmGetIndTermKeyMsg message.

If you receive a tmClickMsg message, the application has received a mouse-down event in the terminal emulation window. Your tool must support placing and dragging the cursor in this window, and it must also call the application's click-loop procedure.

▶ Programming with the File Transfer Manager

The File Transfer Manager provides a standard API to file transfer services that is independent of file transfer protocols, which are implemented in file transfer tools. These tools are described in the following section. File transfer tools bear a similar relationship to the File Transfer Manager that printer drivers have with the Print Manager (Figure 13-3 illustrates this analogy with the Connection Manager). An application calls the Print Manager, which performs relatively little work and then calls the printer driver selected by the user to do the work of printing. In a similar way, an application calls the File Transfer Manager, which performs relatively lit-

tle work. The File Transfer Manager calls the file transfer tool selected by the user to transfer files. Figure 13-9 shows the dialog presented by the XMODEM file transfer tool.

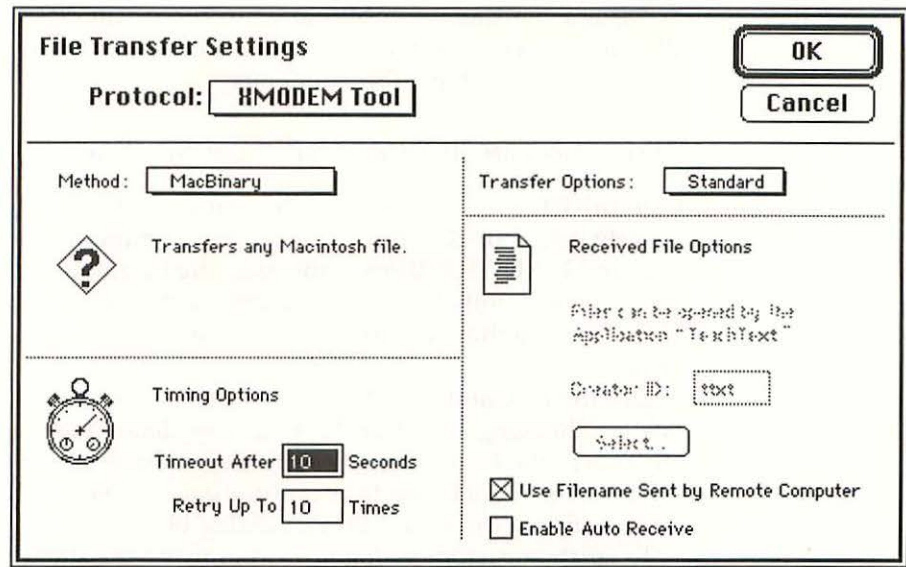


Figure 13-9. Dialog presented by the XMODEM file transfer tool

You can transfer files over more than one connection at a time using the File Transfer Manager. You can use the same or more than one file transfer tool to accomplish this. Also, because of the architecture of the Communications Toolbox, more than one application can use the same tool at the same time.

▶ The File Transfer Record

Applications pass data to the routines of the File Transfer Manager using a protocol-independent *file transfer record*, which contains all the details required to transfer a file, such as the direction of the transfer (to, from, or full duplex), pointers to your procedures to read from and write to disk files, and pointers to input/output buffers. Also, this record contains configuration records for the selected file transfer tool to store its private data. The file transfer record is private to the File Transfer Manager, which provides a set of routines for modifying the contents of the record.

If you need to transfer more than one file over the same connection,

you have to do some more work. You need to create one file transfer record for each file to be transferred; when one file has been transferred, you move to the next. Your code will have to ensure that the connection doesn't have to be restarted for each file. The other way to handle this is to write a file transfer tool that can present the user with a dialog box allowing selection of more than one file. The file transfer tools provided by Apple do not have this capability.

▶ Using the File Transfer Manager Routines

Call **InitFT** to initialize the File Transfer Manager. You should have already called **InitCRM** to initialize the Communications Resource Manager and **InitCTBUtilities** to initialize the Communications Toolbox Utilities. You must initialize these two managers before using the File Transfer Manager, whether you use any of the routines in the latter two managers or not.

Call the **FTChoose** routine to present the user with the standard dialog box for choosing a file transfer tool. This dialog searches for all file transfer tools in the Communications Folder in the System Folder, and the user selects which tool to use from a pop-up menu on this dialog. The user can also configure the tool from this dialog box.

To create a custom dialog instead of using the standard dialog, you can use the six routines provided as a part of the File Transfer Manager, which simplify this code to a certain extent. There is one other method of selecting a tool: your application can also select a file transfer tool by using a scripting language. In this case, the user wouldn't see a dialog box.

To interface with a scripting language, use the two routines provided as part of the File Transfer Manager that simplify this process: **FTGetConfig** and **FTSetConfig**. These routines get a configuration string from and set the configuration string for a file transfer tool. Remember that these calls are being processed by the file transfer tool—the File Transfer Manager does nothing more than provide an API to the tool with these calls.

▶ Preparing to Transfer a File

Use the **FTNew** routine to create a new file transfer record to prepare to transfer a file. When making this call, pass the address of the four procedures in your code that will send data, receive data, read data from a file, and write data to a file. The routines that send and receive data can use the Connection Manager to perform their function. You also need to pass a pointer to a procedure that the file transfer tool can call to find out what environment it's running in, such as how wide the data channel is in bits. Last, you need to

pass the procID of the file transfer tool. Get this value by calling **FTGet-ProcID**. You need to specify the name of the file transfer tool to get its procID. These IDs are assigned dynamically by the File Transfer Manager.

As part of processing the **FTNew** call, the File Transfer Manager calls **FTDefault** to fill in the configuration records in the file transfer record. The File Transfer Manager also loads in the main procedure of the file transfer tool (see the next section for details on file transfer tools).

Use the **FTDefault** routine to ask the current file transfer tool to (allocate and) set the file transfer record to default values. Use the **FTValidate** routine to verify that the file transfer record is internally consistent.

▶ Performing the File Transfer

Use the **FTStart** routine to begin the process of transferring a file. This routine opens the file that will be transferred. You can optionally cause a status dialog to be displayed. Call the **FTExec** routine each time your code passes through the main loop. This routine performs the basic task of the File Transfer Manager—transferring the file. Each time you call **FTExec**, it processes some of the file. When the file has been successfully transferred, the file transfer tool closes the file it was working on and deallocates any memory that it allocated.

If you need to stop the file transfer process, call **FTAbort**. This routine tells the other end of the connection that you are aborting the transfer.

Call **FTDispose** when you're all done with the file transfer process. This routine deallocates the file transfer record and the data structures to which it refers.

▶ Handling Events with the File Transfer Manager Routines

Use the **FTEvent** routine to tell the file transfer tool that the event your application just received happened in a window owned by the tool. The tool should therefore process the event.

Call the **FTMenu** routine to tell the file transfer tool that the menu event your application just received happened in a menu owned by the tool. The tool should therefore process the event.

Use the **FTActivate** routine to tell the file transfer tool that your application has received an activate or deactivate event. The tool may need to do something when this happens, such as installing or removing a menu.

To tell the file transfer tool that your application has received a suspend or resume event, call the **FTResume** routine. When your application is going to be running in the background (or returning to the foreground), the tool may adjust some of its parameters.

▶ Other File Transfer Manager Routines

Call **FTGetToolName** to get the name of the specified file transfer tool. If you want to save the file transfer tool's configuration, you also need to save the name of the tool.

Use the **FTGetRefCon** and **FTSetRefCon** routines to get and set the reference constant in the file transfer record. Use the **FTGetUserData** and **FTSetUserData** routines to get and set the user data (it's 4 bytes, so it can be a handle) in the file transfer record.

To get the 'vers' information from the file transfer tool, call **FTGetVersion**. This information is returned in a handle, which you must dispose of when you're done with it.

▶ Internationalization and the File Transfer Manager

To help internationalize your application, use **FTIntlToEnglish** and **FTEnglishToIntl**, which ask the file transfer tool to translate configuration strings from and to other languages. These routines assume that the strings are stored in the tool in American English.

▶ Writing a File Transfer Tool

A file transfer tool provides a file transfer capability using a particular file transfer protocol, such as XModem or Kermit. File transfer tools have a structure that makes them interchangeable from the user's (and application's) point of view. That is, a user can easily switch from one file transfer tool to another by selecting another file transfer tool. This can be done as easily as choosing another printer.

File transfer tools are structured as a set of resources in a single file. A file transfer tool has a basic set of five or six resources.

- 'fdef' code resource—Implements the primary file transfer functions of the tool.
- 'fval' code resource—Validates file transfer records when the File Transfer Manager is called using the **FMValidate** call. This code also handles the **FMDefault** call.
- 'fset' code resource—Handles the file transfer settings dialog.
- 'fscr' code resource—Handles the interface between a scripting language and the tool.
- 'floc' code resource—Translates this tool's strings between English and other languages.
- 'fbnd' bundle resource (optional)—Contains the tool's name and lists all the resources in this tool.

All of the code resources ('fdef', 'fval', 'fset', 'fscr', and 'floc') are called by the File Transfer Manager. The File Transfer Manager passes along a message, which is a code telling the code resource what to do, and a set of three parameters. The meaning of the parameters is determined by the message, and not all parameters are used for all messages.

The 'fval', 'fset', 'fscr', 'floc', and 'fbnd' resources are identical in function and structure to the 'cval', 'cset', 'cscr', 'cloc', and 'cbnd' resources described previously in this chapter, in the section entitled "Writing a Connection Tool." The prefix for the names of messages, error codes, and data structures associated with these resources is *fm* rather than *cm*. For the details on creating these resources, refer to the previous section. These are not the only resources that are in a file transfer tool; also needed are resources for dialog boxes and dialog item lists managed by this tool. Let's now look at the details of creating an 'fdef' resource.

▶ Creating an 'fdef' Resource

An 'fdef' resource performs the primary functions of the file transfer tool: transferring a file to or from this machine using a particular protocol. A terminal tool might be called by the File Transfer Manager with any of the messages listed in Table 13-3. If your tool does not understand the message or cannot support it, then it should return the error message fmNotSupported. Note that the File Transfer Manager's caller could be either an application or another communications tool.

Table 13-3. Messages for the 'fdef' resource of a file transfer tool

Message Name	*Sent to the File Transfer Tool When the Caller:*
fmInitMsg	Initializes the file transfer
fmDisposeMsg	Closes the file transfer
fmSuspendMsg	Requires the tool to handle a suspend event
fmResumeMsg	Requires the tool to handle a resume event
fmMenuMsg	Got a menu event for a menu belonging to this tool
fmEventMsg	Got a window event and this event is associated with this tool
fmActivateMsg	Requires the tool to handle an activate event
fmDeactivateMsg	Requires the tool to handle a deactivate event
fmAbortMsg	Wants to abort a file transfer
fmStartMsg	Wants to start transferring a file
fmExecMsg	Wants to provide time to transfer the file

You'll now look at several of the basic messages every terminal tool should support: fmInitMsg and fmDisposeMsg, fmStartMsg, fmExecMsg, and fmAbortMsg.

The fmInitMsg message tells your code to initialize for a new file transfer session. This is the time to allocate any local memory for this tool, including buffers. The fmDisposeMsg message tells your code to deallocate any local memory that was allocated when you received the fmInitMsg. The File Transfer Manager handles the deallocation of the configuration records and the file transfer record. If you try to deallocate them, you'll cause a system crash.

The fmStartMsg message tells your code to begin transferring a file. Your code should open the file to be transferred, prepare the connection, and draw a status dialog.

The fmExecMsg message gives this tool some time to transfer the file. This message is received repeatedly until the transfer has been completed. When it completes, close all files and remove the status dialog.

The fmAbortMsg message tells your tool to halt the file transfer. You also need to close all files and remove the status dialog.

▶ Programming with the Communications Resource Manager

The Communications Resource Manager provides routines for handling resources and devices. These routines make your job easier by arbitrating between applications using the Communications Toolbox.

Call the **InitCRM** routine to initialize the Communications Resource Manager. You must do this before making any other call to any of the other managers in the Communications Toolbox. Also, you must make this call after making all the standard toolbox initialization calls.

Communications devices are managed by means of a queue. This queue is maintained by the Communications Resource Manager. Each entry in the queue describes a communications device, including the device type, the device ID, attributes, and status.

Use the **CRMInstall** call to install a device into the queue. You need to fill out a record for the queue before calling this routine. Typically, a driver or INIT would do this rather than an application. This is an important call, because the other managers in the Communications Toolbox can only use devices that have been registered with the Communications Resource Manager. Each Macintosh has two built-in serial ports, so at a minimum there should be two records in the queue. Use the **CRMRemove** call to remove a device from the queue.

Use the **CRMSearch** routine to search through the queue for a device of the specified type and with an ID greater than the specified ID. You might use this routine in an application to find a specialized type of communications device, such as a serial port that can support high transmission rates.

The Communications Resource Manager provides a series of routines to get and release resources. These routines provide one additional service beyond the Resource Manager routines that they call: the Communications Resource Manager maintains a use count on each resource. Each time a resource is obtained, its use count is incremented. Each time a resource is released through a Communications Resource Manager routine, its use count is decremented. Only when the use count reaches zero is the resource really released. At this point, the Communications Resource Manager calls the equivalent Resource Manager routine to accomplish this. Thus, you don't have to worry about another application using the same communications resources that you do. The Communications Resource Manager handles all the bookkeeping for you.

The Communications Resource Manager provides the following routines: **CRMGetResource**, **CRMGet1Resource**, **CRMGetIndResource**, **CRMGet1IndResource**, **CRMGetNamedResource**, **CRMGet1NamedResource**, and **CRMReleaseResource**. Each of these routines behaves like the Resource Manager routine it is named after, except that the Communications Resource Manager automatically maintains a use count on the resource.

Use the **CRMGetIndex** routine to find out what the usage count is for a particular resource. Use the **CRMGetIndToolName** routine to find out the name of a particular communications tool. Last, use the **CRMRealToLocalID** and **CRMLocalToRealID** routines to map between physical resource IDs (that is, the resource IDs used in the communications tool file) and local IDs (that is, the IDs for the resources in memory).

▶ Programming with the Communications Toolbox Utilities

Initialize the Communications Toolbox Utilities after initializing the Communications Resource Manager by calling **InitCTBUtilities**.

The **NewControl** call provides you with a pop-up control procedure (CDEF), which is part of the Communications Toolbox. You can use this procedure to give users control over baud rate, parity, and other options when writing terminal, file transfer, or connection tools. You can also use this CDEF in other parts of your application; you aren't restricted to using it just for communications tools.

Use the **AppendDITL**, **ShortenDITL**, and **CountDITL** calls to append, remove, or count dialog items to an existing dialog box. You can use these routines on any dialog box, not just those associated with the Communications Toolbox.

In fact, starting with System 7, these routines now belong to the Dialog Manager. Before using any of these three new routines, verify that they are available by calling **Gestalt** with a selector of gestaltDITLExtAttr. This call will return the attributes of the Dialog Manager extensions. At the release of System 7, the only attribute returned is that these three routines are available or not available.

Use the **NuLookup** routine to provide a dialog box with a scrollable list of AppleTalk entities for your users. This dialog is similar to that presented by the Chooser when there are zones on the current network. In this case, the user can scroll through a list of zones and then through a list of entities in that zone.

You can specify a list of one or more types of entities to be displayed. The code for the dialog box makes a call using the Name Binding Protocol to find entities of those types. You can optionally supply two filter procedures: One protocol can filter out object, type, and zone tuples from the list of entities to be displayed; the other can filter out zones from the dialog's list of zones. When the routine returns, you're given a value specifying whether the user selected an object. If the user did select an object, then you're also given its name and network address.

By the Way ▶ A **tuple** is an ordered list of *n* elements.

If you need even more control over the appearance of the dialog box displaying the network entities, use the **NuPLookup** routine. You must pass this routine a dialog ID, which must have a set of dialog items corresponding to, but not necessarily the same as, the standard dialog presented by the **NuLookup** routine. In addition to changing the appearance of these items, you can add other items to the dialog box.

▶ Conclusion

In this chapter, you've looked at the Communications Toolbox. This toolbox consists of five managers: the Connection Manager, the Terminal Manager, the File Transfer Manager, the Communications Resource Manager, and the Communications Toolbox Utilities. These managers call communications tools: connection tools, terminal tools, and file transfer tools.

Collectively, these managers make the job of adding communications and networking to an application much easier than it would be if you had to do it from scratch. Users find that the Communications Toolbox has also made their lives easier by providing standard interfaces for setting up and using communications links.

Get Info ▶

For more information on the Communications Toolbox, don't look in *Inside Macintosh*—it's not there. Instead, the toolbox is described in a set of documents available from APDA: the *Macintosh Communications Toolbox Reference*, the *Macintosh Communications Toolbox Source Code Examples*, and the *Communications Tools Basic Connectivity Set*. The first manual details how to call routines in the Toolbox, how to call communications tools from applications, and how to write communications tools. The second publication provides several examples in the form of source code. The third publication documents the basic set of communications tools, including Teletype, VT102, and VT320 terminal emulators; serial and modem connection tools; and text and XMODEM file transfer tools.

14 ▶ AppleTalk Phase II and AppleShare

▶ Introduction

AppleTalk has provided a simple, easy-to-install local area network that has been built into every Macintosh ever shipped. At first, AppleTalk was used primarily as a way of sharing a relatively expensive LaserWriter. Gradually, though, other peripherals were developed to be shared over AppleTalk. Shareable devices available today include gateways to other networks, modems, serial ports, and file servers. Multiuser applications such as databases are also available. In this chapter, you'll first briefly review the AppleTalk protocol architecture and the drivers that implement them.

AppleTalk Phase II, introduced in 1989, enhanced the AppleTalk architecture by improving and adding protocols and reducing some of the limitations of the earlier implementations. AppleTalk Phase II is shipped as part of System 7. This is the second topic that you'll explore in this chapter.

The third topic of this chapter will be a new protocol, AppleTalk Data Stream Protocol (ADSP), which was introduced prior to System 7 and AppleTalk Phase II. ADSP, a standard protocol provided as part of System 7, is the first AppleTalk protocol that application programmers should think of using.

The final topic of this chapter is File Sharing, a personal version of the AppleShare file server software. It provides, for the first time, a standard file server technology as part of the operating system. Previously, application developers could not assume the existence of a standard file server, so developers of multiuser software had to ask their users to purchase file

server software (and sometimes hardware). There was (and still is) an alternative: developing application-specific network protocols. This is not an easy task (if it's done correctly), and so few developers ever went this route.

File Sharing allows application developers to assume that a file server is readily available from all Macintosh computers. This will encourage the development and use of multiuser software.

▶ AppleTalk Protocols and Drivers

The AppleTalk protocol suite is illustrated in Figure 14-1. Refer to *Inside AppleTalk* for information about these protocols (see the "Get Info" section at the end of this chapter). Most of these protocols are of interest primarily to programmers developing low-level network software, such as the code in a network bridge. Notice that the protocols are layered. One important implication of this is that a higher-level protocol assumes the existence of services provided by the lower-level protocols under it.

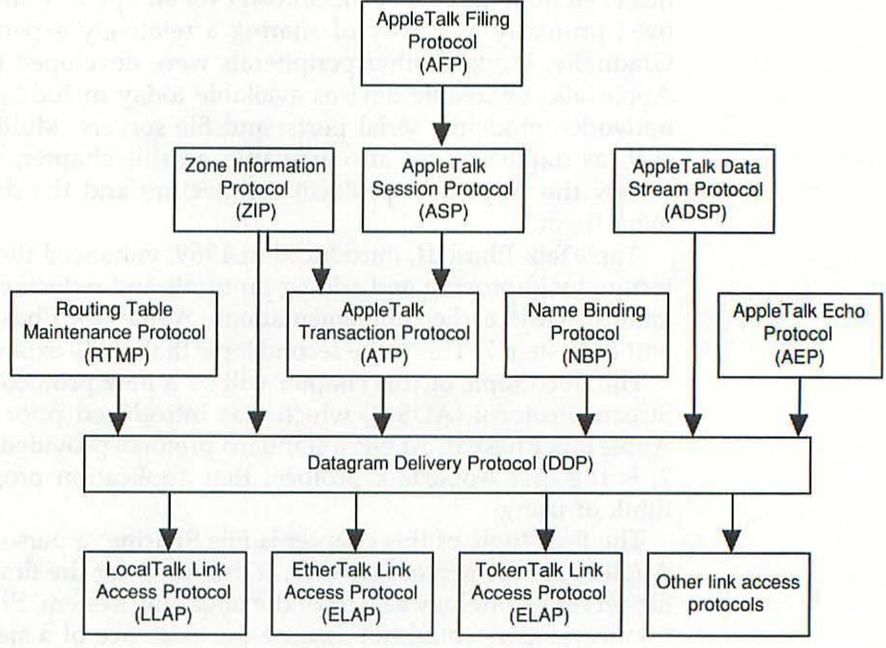


Figure 14-1. The AppleTalk protocol suite

By the Way ▶

A few words on terminology: *AppleTalk* is the name of the protocol architecture. It used to be the name of the networking implementation that is built into every Macintosh computer; that implementation of the protocol is now known as *LocalTalk*. This change in terminology was made when *EtherTalk* was introduced. EtherTalk is an implementation of the upper layers of the AppleTalk protocol suite on top of Ethernet. More recently, Apple has introduced *TokenTalk*, which is an implementation of the upper layers of the AppleTalk protocol suite on top of *Token Ring*. Both Ethernet and Token Ring networks are commonly used to connect workstations, minicomputers, and mainframes.

The protocols are implemented in several different drivers that make up the AppleTalk software, described in Table 14-1. The first four drivers are provided as part of the standard system in System 7. The Ethernet driver is provided with Apple's Ethernet card.

Table 14-1. AppleTalk drivers and protocols

Driver Name	*Protocols Implemented*
.MPP	LLAP, DDP, NBP, AEP, and RTMP stub
.ATP	ATP
.XPP	ASP and workstation portions of ZIP and AFP
.DSP	ADSP
.ENET	Ethernet driver

Note ▶

The one major omission in the AppleTalk architecture is network management protocols. These protocols, which would be implemented in all network-related hardware and software (but not in application software), would make it much easier to locate and isolate network problems. These protocols would also enable developers to tune a network for optimal performance. Apple has indicated that these protocols are under development.

At the 1990 Worldwide Developer's Conference, Apple announced that it would license various portions of its AppleTalk code and protocol testing tools for porting to other platforms. Some components can be licensed in object code only; others can be licensed as either source code

or object code. Apple did this for several reasons. First, it will encourage more AppleTalk products to be developed because implementing network protocols is a difficult task, and licensing existing code eliminates most of that work. Second, it means that products developed using this licensed code will be more compatible with existing AppleTalk products than products developed without this code.

▶ Protocols of Interest to Application Programmers

Three protocols of interest to application developers that can be used as a foundation for building multiuser cooperative applications are AppleTalk Transaction Protocol (ATP), AppleTalk Session Protocol (ASP), and AppleTalk Data Stream Protocol (ADSP).

ATP, a lower-level protocol than the other two protocols discussed here, is useful for sending small amounts of data from one socket to another, which in practice usually means from one machine to another. Because ATP does not provide any support for sessions, there is little overhead to send a small amount of data, but it is too cumbersome for handling large amounts of data or for exchanging many packets of data.

ASP is a protocol designed to support clients and servers. One end of the ASP protocol is for clients of a service, and the other end is for the providers of that service. The AppleTalk Filing Protocol (AFP) is built on top of ASP. Only the client portion of ASP is provided as part of the standard AppleTalk software shipped with the Macintosh system.

ADSP becomes a standard protocol starting with System 7, although it was available previously from APDA. ADSP provides a protocol for communicating between two equal entities (known as a peer-to-peer protocol). You can also use ADSP to implement clients and servers. Unless you have good reasons for using either of the previous protocols, ADSP is the protocol of choice. This protocol is discussed later in this chapter.

▶ AppleTalk Phase II

AppleTalk Phase II brought the first substantial revision in the architecture of the AppleTalk protocol suite and in Apple's implementations of it. In this section, you'll first look at a summary of the changes of primary interest to application developers. Look in the "Get Info" section at the end of this chapter for references to complete documentation on the AppleTalk architecture.

The goals of AppleTalk Phase II are to allow AppleTalk networks to do the following:

- Support more than 254 nodes per network
- Provide better support for *internets* (a group of connected networks)
- Remain compatible with current AppleTalk applications and yet provide extended capabilities for future products
- Require no changes to nonrouting LocalTalk nodes

Upgrading a network to Phase II will not affect machines using LocalTalk. On the other hand, machines with EtherTalk cards (for use on Ethernet networks) and TokenTalk cards (for use with Token Ring networks) will be affected much more.

▶ AppleTalk Phase II Features for Application Developers

AppleTalk Phase II has brought substantial changes to the AppleTalk protocol suite and to Apple's implementations of it. Many of the features introduced in Phase II are of interest only to hardware and system software developers. For the most part, these changes will not be directly visible either to application programmers or to users. For example, the Routing Table Maintenance Protocol now uses a technique known as *split horizon*, which reduces the size of broadcasts when nodes using this protocol exchange their network maps. See the references listed at the end of this chapter for more information.

Phase II includes the following features of interest to application developers:

- Improvements to the AppleTalk Transaction Protocol (ATP), which provide for several values for the release timer and a call to cancel all pending asynchronous calls for a particular socket.
- Improvements to the Zone Information Protocol (ZIP), which now permits a single physical network to contain several zones. Several calls have been added to make it easier for applications to obtain zone information.
- An AppleTalk Transition Queue has been added to the .MPP driver to arbitrate between applications using the AppleTalk drivers.
- A new call to the .MPP returns network status information about the current node.
- Another wildcard character has been added for use in requesting names using the Name Binding Protocol (NBP).

You'll look at each of these new features in turn.

▶ Improvements in the AppleTalk Transaction Protocol

One call has been added to support the ATP, and one call has been enhanced.

The **ATPKillAllGetReq** call cancels all pending asynchronous calls made through the **ATPGetRequest** call for the specified socket. This call does not close the specified socket.

The **PSendResponse** call now supports five different values for the timeout value for exactly-once transactions. Previously, the timeout interval was always 30 seconds in this case. With Phase II, the additional intervals of 1, 2, 4, and 8 minutes are supported. These longer values will help when applications are running on large internets.

▶ Improvements to the Zone Information Protocol

One of the most important changes to the Zone Information Protocol is that a single physical network (except LocalTalk networks) can now contain more than one zone. Such a network is called an *extended network*. This means the limit of 254 nodes per network (except on LocalTalk networks) becomes approximately 16 million.

Before Phase II, you'd have to use ATP to query the routers on your internet to obtain a list of zones. Three new routines in the .XPP driver make it easier to get zone information. All three calls use the new xCallParam parameter block, which is described in detail in the AppleTalk Manager chapter of *Inside Macintosh*, Volume VI. You must fill out parts of the parameter block before making any of the three calls, and the parameter block is returned with the requested information.

Use the **GetMyZone** call to find the name of the zone in which the current node resides. Use the **GetLocalZones** call to get a list of all the zone names on the local network. This list will always have only one entry if the local network is LocalTalk. Use the **GetZoneList** call to get a complete list of all the zones on the internet.

▶ The AppleTalk Transition Queue

More than one application can use the AppleTalk drivers concurrently. If one application closes the .MPP driver while another application is using it, the second application would have problems dealing with it. The AppleTalk Transition Queue was added in Phase II to arbitrate between applications.

The AppleTalk Transition Queue is managed by the LAP (Link Access Protocol) Manager, a low-level manager that is rarely, if ever, used by

applications. The queue is handled by this manager because, unlike the drivers, the LAP Manager is always running.

Opening the .MPP driver is unlikely to affect other applications that are already using the driver. When a request is made to close the .MPP driver, the system walks through the queue and checks with each application listed in the queue to find any objections to closing the driver. Anyone can object at this time.

If there are no objections to closing the driver, then the system closes the driver. If there are any objections, the system displays the name of the application that wants to continue using the driver. The user can choose to force the driver to close, and if this happens, the system walks through the queue and tells each application to shut down any AppleTalk connections. No objections are allowed at this time.

▶ Using the AppleTalk Transition Queue

Use the **LAPAddATQ** call to add an entry to the AppleTalk Transition Queue. The AppleTalk Manager will call the routine whose address is in the entry when any software has done one of the following:

- Open the .MPP driver
- Close the .MPP driver
- Call the **PATalkClosePrep** function
- Deny another routine in the queue permission to close AppleTalk
- Call **ATEvent** or **ATPreFlightEvent** routine

These five situations are called AppleTalk *transitions*.

Use the **LAPRmvATQ** call to remove an entry from the AppleTalk Transition Queue.

Generally applications should not close the .MPP driver since another application may be using it. However the system may, at certain times, want to close this driver. It calls the **PATalkClosePrep** routine in the .MPP driver before doing so, passing a selector code. This routine then checks with each routine in the AppleTalk transition queue for permission to close the driver. Each transition queue routine uses the selector code to tell it what type of transition has been requested (or commanded, since some transitions are mandatory).

You can define your own AppleTalk transition and define your own selector code for it. Rather than using **PATalkClosePrep** to check with the AppleTalk transition queue routines, you will use either **ATEvent** or

ATPreFlightEvent system calls. Call **ATEvent** with your selector code to notify each queue routine of your transition. Call **ATPreFlightEvent** with your selector code to notify each queue routine of your transition. You will also pass a second selector code which will be passed to each of the queue routines should any of them return a non-zero result after receiving the first selector.

Your AppleTalk transition queue routine should return a result of zero for any selector which you choose not to handle or do not know about. Returning a non-zero result in either of these cases may cause problems for either the system software or for some other application.

▶ Obtaining Current Node Information Using the .MPP Driver

Use **PGetAppleTalkInfo**, a new call to the .MPP driver, to get the following information about the current machine (node):

- Whether the capability of sending packets from the current node to itself is enabled
- The low and high values of network numbers on the local cable (if the cable is not LocalTalk) or, if the cable is LocalTalk, the network number
- The 24-bit AppleTalk network address for the current node
- The 24-bit AppleTalk network address for the last router that the current node has heard from

Other information is also returned in this parameter block. This call makes it much easier and simpler to find out about the current node and its status on the network.

▶ Wildcard Characters for the Name Binding Protocol

Applications use the Name Binding Protocol primarily to look up addresses on the network or internet. A common reason for doing this is to locate a server. When requesting addresses, you can specify whether you want the search to take place in the current zone or all zones.

You can now use three wildcard characters when requesting network addresses. An equal sign (=) represents all possible non-null values for the object or type fields. An asterisk (*) represents the current zone, so that you don't have to look up the current zone name. Phase II adds the tilde (~), which you can use to match any or no characters in the object or type fields. Only nodes running Phase II drivers will recognize this wild-

card character. An example of using the tilde is *a~e*, which would match Apple, Ale, and Age.

▶ Compatibility and AppleTalk

To use the new functions described in this chapter, you need to find out which version of the AppleTalk drivers you're running with. Do this by calling **Gestalt** with a selector of gestaltAppleTalkVersion. If the version number is greater than or equal to 53, then your driver implements Phase II.

▶ AppleTalk Data Stream Protocol

The AppleTalk Data Stream Protocol (ADSP) provides a robust, easy-to-use protocol that application programmers can easily use. You can transmit a stream of bytes (hence the protocol name) from one node to another. The protocol divides the stream into a set of packets and automatically reassembles them at the other end. This makes it easy for applications to talk to one another.

After you open a connection between any two nodes on an internet, they can exchange data using ADSP. This means that the nodes can be on a single network or on different networks. Only one connection can be open between any pair of nodes, but any node can have open connections to other nodes.

▶ Using the ADSP—Important Data Structures

To use ADSP, you first have to use the **MPPOpen** call to open the .MPP driver. This driver handles the lower-level protocols that ADSP runs above. You then need to call **OpenDriver** to open the .DSP driver, which implements ADSP.

To establish and use a connection over ADSP, you need to allocate a *Connection Control Block* (CCB), which is described in detail in the AppleTalk Manager chapter of *Inside Macintosh*, Volume VI. This data structure is used by ADSP for storing its internal variables. You can read from the fields in a CCB, but for the most part, you are not allowed to change anything in it.

To use ADSP, you first fill out a DSPParamBlock record, which is described in the AppleTalk Manager chapter of *Inside Macintosh*, Volume VI. After filling out this data structure, you then make the appropriate system call. Calls are handled by the .DSP driver, which is shipped as part of System 7.

▶ The ADSP Calls

Let'll now look at each ADSP-related call in turn and see what it is used for. Following this, you'll look at how to use them.

Use the **dspInit** call to assign a socket for use by the .DSP driver and initialize the variables needed for this connection. This call does not open the connection, but prepares everything for the open call. You need to pass the address of the CCB for this connection. A CCB reference number is returned if this call is successful, and you'll use this reference number in all other calls to the .DSP driver.

Call **dspRemove** when you are done with a connection (after the connection has been closed).

To open the connection, use the **dspOpen** call. You can choose from one of four connection modes.

- ocRequest mode—Attempts to open a connection with the specified network address for the remote node
- ocPassive mode—Waits for an open-connection request from a remote node
- ocAccept mode—Used by a server to open a connection to a client
- ocEstablish mode—Used by a client to open a connection with a previously known node

To close a connection, call **dspClose**. The connection end still exists, and you can still read from the receive queue.

Use the **dspCLInit** call in place of **dspInit** if you are opening a socket that will be listening on the network. The *CL* in the call name stands for Connection Listener. If you are developing a server using ADSP, the server would open sockets using **dspCLInit**, which functions much like the **dspInit** call.

To listen for connection requests, use the **dspCLListen** call in a server. You must have used the **dspCLInit** call to initialize the connection end. You can call **dspCLListen** several times to accept more than one connection.

To deny a request from a client for a connection, use the **dspCLDeny** call in a server.

Use the **dspCLRemove** call to close a connection end for a server. It functions much like the **dspRemove** call.

To find out the current status of a connection, call **dspStatus**. This call returns the number of bytes remaining to be sent and received, and the space left in the send and receive queues. It also returns a pointer to the

CCB for this connection, which you can use to find other information about the connection.

Use the **dspRead** call to read bytes from the connection's receive queue. You can request that it read a certain number of bytes, and when the call has completed, it will tell you how many bytes it actually read.

Call **dspWrite** to write bytes to the connection's send queue. Bytes remain in the queue until they have been transmitted to the remote node and that node has acknowledged receiving them. The send queue transmits bytes to the remote node when you call **dspWrite** with the flush parameter on, when the number of bytes in the send queue reaches its limit (known as the blocking factor), when the send timer expires, or when an acknowledgment packet is required to be sent to the remote node. You can send an arbitrary amount of data with one call to **dspWrite**. Bytes will be sent a buffer at a time until all the bytes have been sent.

To send an attention message to the remote node on the connection, use the **dspAttention** call. This feature is especially useful for sending control messages or data separate from the data sent with the **dspRead** and **dspWrite** calls.

Use the **dspOptions** call to set optional parameters for the connection, such as the maximum number of bytes that should be accumulated before ADSP sends out a packet. Another parameter is the maximum number of out-of-sequence data packets that the local end can receive before requesting the remote end to retransmit.

To reset and resynchronize the connection, use the **dspReset** call. All data in the send queue, in transit, and in the remote node's receive queue will be discarded.

▶ Using the ADSP Calls

You'll now look at what it takes to open and use an ADSP connection with another node. You can use the calls in this subsection to create an online conferencing application or a multiuser game running over a network. You can also use these calls to talk with a server process, either on the same machine or another machine. In this case, the client of the server would use the calls in this subsection, and the server would use the calls in the following subsection. For details, refer to the AppleTalk Manager chapter of *Inside Macintosh*, Volume VI.

First, open the .MPP driver by calling **MPPOpen**, and open the ADSP driver (.DSP) by calling **OpenDriver**. Then, allocate memory for the Connection Control Block (242 bytes), for the send and receive queues (600 bytes or more per queue), and for an attention message buffer (570 bytes). These blocks of memory must be locked, so you can allocate them using

NewPtr. Remember that this memory is owned by the ADSP driver after you initialize the socket with the **dspInit** call, so you cannot read or write to this memory directly until you shut the connection down with a call to **dspRemove**.

Call **dspInit** to set up the connection end on this node. If you need a specific socket number, you can ask for it at this time. Call **NBPRegister** now if you need to establish the name and address of this socket on the network. Call **dspOptions** if you need to modify any of the default parameters for this connection end.

Call **dspOpen** to open the connection using one of the four modes explained earlier: ocAccept, ocEstablish, ocRequest, or ocPassive. If you are talking with an ADSP listener, you should look up its address using the **NBPLookup** system call and then call **dspOpen** in ocRequest mode.

Use **dspRead** and **dspWrite** to send and receive data with the remote socket. Use **dspAttention** to send an attention command to the remote socket.

Call **dspClose** when you're done talking with the remote node and you want to continue using the local connection end. Call **dspRemove** if you have no further need of the local connection end.

▶ The ADSP Calls for a Connection Listener

Now let's look at what it takes to open and use an ADSP connection listener. For the details, refer to the AppleTalk Manager chapter of *Inside Macintosh*, Volume VI. A *connection listener* is a special kind of connection that exists solely to receive open-connection requests. It hands the request to the connection server, which can establish a connection end and send an acknowledgment back to the node that requested the connection.

First, open the .MPP driver by calling **MPPOpen**, and open the ADSP driver (.DSP) by calling **OpenDriver**. Then allocate memory for the Connection Control Block (242 bytes). This block of memory must be locked, so you can allocate it using **NewPtr**. Remember that this memory is owned by the ADSP driver after you initialize the socket with the **dspCLInit** call, so you cannot read or write to this memory directly until you shut the connection down with a call to **dspRemove**.

Call **dspCLInit** to set up the connection end on this node. If you need a specific socket number, you can ask for it at this time. Call **NBPRegister** now if you need to establish the name and address of this socket on the network.

To wait for an open-connection request, call **dspCLListen** asynchronously. If you made this call synchronously, your application wouldn't be able to do anything until a request was sent. When a request does come

in, you'll be told the remote node's address and parameters associated with that node.

If the server accepts the request, call **dspInit** to set up a local connection end, then call **dspOpen** with a mode of ocAccept. The server and client can now send and receive data using the **dspRead** and **dspWrite** calls, and can send attention messages using the **dspAttention** call. Use **dsp-Close** or **dspRemove** to close the connection when you're done. If the server denies the request, call **dspDeny**. To wait for another open-connection request in either case, call **dspCLListen** again.

When you're ready to shut down the connection listener, call **dspCLRemove**.

▶ File Sharing

As mentioned in the introduction to this chapter, File Sharing provides a personal version of the AppleShare file server software. Thus, programmers can always assume their applications can have access to a file server. Also, users on small networks do not have to purchase an additional machine for use as a file server. Several upgrade paths beyond File Sharing are available, however. Users can purchase an additional Macintosh to run as a dedicated AppleShare server, or they can purchase other third-party file server software that uses the AppleShare protocol.

The client portion of AFP, AppleTalk Filing Protocol, has been a part of the standard AppleTalk software shipped as part of the system for some time. This protocol is used to request files and data from files located on a server. Portions of a file can be locked using byte-range locking, as described in the File Manager chapter of *Inside Macintosh*, Volume IV, without having to deal with AFP directly. Writing multiuser application software becomes a lot easier because the File Manager handles AFP for you.

Client software is also available for the Apple IIe, Apple IIGS, and MS-DOS machines. The Apple IIe and MS-DOS machines require a card to connect them to an AppleTalk network. The server portion of AFP runs on AppleShare servers, and now on any Macintosh that has enabled File Sharing.

In System 7, for the first time a version of the server software that understands AFP is also a standard part of the Macintosh system. File Sharing can be used to share up to ten hard disks (whether they are fixed, removable, or CD-ROMs) or folders. Floppy disks cannot be shared.

Volumes or directories shared using File Sharing behave just like AppleShare volumes on the network. Once an AppleShare or File Sharing volume has been mounted, it behaves like any other mounted volume to the user with the exception of access control.

Each machine has an owner name, password, and Macintosh name. The machine's owner can use his or her owner name and password to access all files on his or her machine from anywhere else on the network. The Macintosh name is used by others to select the machine from the network.

A user can manage File Sharing on his or her machine by specifying the following for each user and for the guest account.

- Which volumes and/or directories will be shared
- What type of access is permitted to each volume or directory (read-only, read/write, and so on)

Each user controls the access to his or her machine by means of the Users and Groups Control Panel, as illustrated in Figure 14-2. The user must create a description of each user who will be permitted access. A special guest account can be used to allow access to anyone without specific permission, as illustrated in Figure 14-3.

Remember that access to a Macintosh (and what remote users have access to) is controlled solely by the local user. This is a potential network management problem if File Sharing becomes widely used beyond local workgroups. The solution to this problem is to migrate files that are used beyond local workgroups to AppleShare servers. AppleShare servers have management tools, which reduce the administrative burden of network and system administrators. The capabilities of File Sharing can be administered locally only.

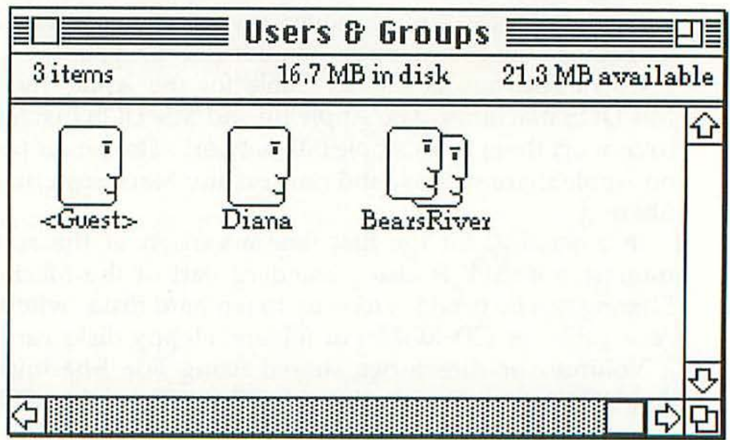


Figure 14-2. The Users and Groups folder

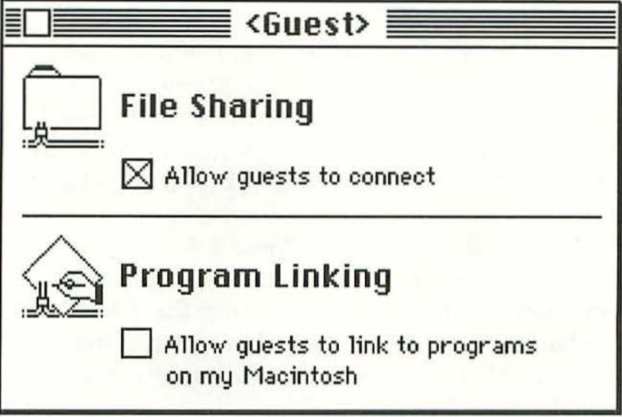


Figure 14-3. Controlling access for the Guest account

If your application potentially depends on the use of File Sharing, the documentation that accompanies the application should describe how users should set up File Sharing on their machines.

A Macintosh running with File Sharing will appear as an AppleShare server to other machines on the same internet. A Macintosh running System 6 will be able to access files on such a machine.

▶ Conclusion

In this chapter, you've looked at AppleTalk Phase II and Macintosh File Sharing.

AppleTalk Phase II has eliminated or reduced many limitations and inconveniences in the previous version of the architecture. Most of these changes are, fortunately, transparent to application developers and users.

The AppleTalk Data Stream Protocol provides a powerful, but easy-to-use, protocol for application developers. ADSP can be used for both peer-to-peer communications and client-server systems.

Macintosh File Sharing provides a low-cost file server that application programmers can now assume is omnipresent on Macintosh networks. This reduces one large barrier to multiuser software—the availability of an inexpensive file server.

Get Info ▶

For more information about AppleTalk Phase II, read the AppleTalk Manager chapter of *Inside Macintosh*, Volume VI. For more information about the AppleTalk Manager, read the AppleTalk Manager chapters of *Inside Macintosh*, Volumes II, IV, and V. For a detailed explanation of the AppleTalk architecture and each of the protocols, refer to *Inside AppleTalk* (second edition, Addison-Wesley, 1989). Other useful references are available through APDA, including *AppleTalk Phase 2 Protocol Specification* and *Macintosh AppleTalk Connections Reference*.

For more information about using the File Manager for multiuser File Sharing, refer to the File Manager chapter of *Inside Macintosh*, Volume I. Also read *Software Applications in a Shared Environment*, available from APDA.

To get more information about licensing the AppleTalk code from Apple, contact the Apple Software Licensing Department at 408/974-4667.

15 ▶ QuickDraw

▶ Introduction

In this chapter, you will look at the managers associated with QuickDraw, including the latest version of QuickDraw itself. The other managers covered in this chapter are the Color Picker Package, the Palette Manager, the Picture Utilities Package, and the Graphics Device Manager.

▶ Some Background on QuickDraw

QuickDraw is the name of the graphics primitives in the Macintosh operating system. All other managers of the operating system and all applications use QuickDraw directly or indirectly to draw on the screen and to output devices (such as printers).

All of QuickDraw's functions are ultimately performed by a set of thirteen procedures known as the QuickDraw bottleneck procedures. Supporting a new screen or output device comes down to writing a new set of QuickDraw bottleneck procedures, and this clearly works strongly to the advantage of application programmers. Without exaggerating too much, programmers writing applications can avoid worrying about which screens and printers their users have installed. Apple has forced the screen and printer manufacturers to support their devices so that application programmers don't have to. This is different from most other operating systems, such as MS-DOS, where each application must contain code to handle each monitor and printer—an onerous burden for application programmers.

Of course, if you want to take advantage of advanced features, such as

32-bit direct colors, you'll have to write some code to use it. If, on the other hand, color is not an important feature, your application can still be used with any screen and any monitor.

QuickDraw shipped with the first Macintosh. Color QuickDraw, which supports 8-bit indexed color, shipped with the Macintosh II. The third version, 32-bit QuickDraw, was first shipped with System 6.0.3 in May, 1989. With System 7, Apple is shipping an enhanced version of 32-bit QuickDraw.

▶ 32-bit QuickDraw

With System 7, 32-bit QuickDraw has been improved and integrated into the operating system. This is the third major version of QuickDraw.

▶ Direct Color Devices

The biggest change from Color QuickDraw is that direct color devices are now supported. In an indexed device (typically a color screen), colors are specified directly by giving a value for the red, green, and blue components of the color (hence RGB). RGB colors are specified (in the Macintosh operating system) as a set of three 16-bit integers, so an RGB color is therefore specified as a 48-bit number. The color device displays this color to the precision specified by the user in the Monitor cdev. This precision is obviously limited by the characteristics of the device.

The latest version of Color QuickDraw supports two depths for use with direct color: 16-bit and 32-bit. Each pixel in a bitmap (color bitmaps are known as *PixMaps*) requires 16 bits or 32 bits of memory. This obviously requires a lot of RAM.

Since each of the three components of a color must be specified with the same number of bits, 16-bit direct color really means that 5 bits are used to specify the value of each component with 1 bit left over.

Similarly, 32-bit direct color means that 8 bits are used to specify the value of each of the three RGB components. You can use the 8 bits left over for your own purposes. Why 8 bits and not 10 bits? The machine instructions of the Motorola family of CPU chips can work directly with 8-bit bytes, but can't work easily with 10-bit integers.

The primary advantage of direct color devices is that the colors in an image can be represented extremely accurately. In fact, there are more colors available in 32-bit mode than the human eye can distinguish. Direct color devices offer photographic-quality color that indexed color devices can only approximate.

▶ Indexed Color Devices

Let's take a brief look at indexed devices to better understand both types of devices. With an indexed device, instead of specifying colors directly, you specify the color of a pixel by giving an index number into a palette of colors. The device maintains this palette in a Color Lookup Table (CLUT). Palettes can typically hold no more than 256 colors, but each color is specified as a 48-bit RGB color. So you cannot display a lot of colors at one time on the screen, but you have a lot of colors to choose from.

One advantage of indexed devices is that you can animate colors easily because you are only changing the values in the color lookup table, not the values in the bitmap. It doesn't take much time to change 256 numbers, so color animation using this technique has good performance.

Another advantage is that less memory is required for indexed devices than for direct devices. QuickDraw supports 1-, 2-, 4-, and 8-bit indexed devices. Therefore, a byte (at most) is needed to specify the color of an individual pixel. The memory requirements for an indexed device are therefore less than for a direct device, and substantially less compared to 32-bit direct devices.

▶ Improvements in 32-bit QuickDraw

So far, you have looked at the differences between 32-bit QuickDraw and the two preceding versions of QuickDraw. Let's discuss the improvements that have been made to 32-bit QuickDraw now that it's a part of System 7.

The changes made to 32-bit QuickDraw are as follows:

- Support for direct (as opposed to indexed) color in PixMaps and the PICT2 format
- Several new QuickDraw calls
- Improved support for gray-scale displays
- New routines that make it possible to tell QuickDraw you've been modifying its data structures

Let's cover each of these changes in turn.

▶ Support for Direct Color in PixMaps

A *PixMap* is the data structure used to hold all the information about a color picture. PixMaps, which were introduced with the first version of Color QuickDraw, replaced BitMaps in Classic QuickDraw.

Note ▶ A BitMap is always one bit deep, that is, it can only represent a black-and-white image. A PixMap can be up to 32 bits deep and can be used to represent black and white, gray scale, or color images.

Several changes have been made to the structure of PixMaps to support direct color. These changes, which do not change the size of the PixMap data structure, are changes in how certain fields in the data structure should be interpreted. The six fields that have changed are as follows.

- pixelType—Can now have a value of RGBDirect, which means that the PixMap contains direct, and not indexed, color.
- pixelSize—Can now be 16 or 32. Previously, the only valid values were 1, 2, 4, or 8. In all cases, pixelSize must be a power of 2.
- cmpCount—The number of components in each pixel. For direct color PixMaps, cmpCount is 3, whereas for indexed PixMaps, it is 1.
- cmpSize—The size of each component in a pixel. For direct devices, cmpSize can be 5 (for 16-bit PixMaps) or 8 (for 32-bit PixMaps). For indexed PixMaps, this number is equal to the pixelSize field. The cmpSize is equal for all components of a color, and cmpSize * cmpCount must be less than or equal to pixelSize.
- rowBytes—Previously had to be less than $2000, but can now be larger: rowBytes must now be less than $4000. For optimum performance, rowBytes should be a multiple of 4, but in any case it must be an even number.
- pmVersion—The version number of the PixMap data structure can be 4, in which case the PixMap's base address is assumed to be 32-bit clean. This field is normally used by low-level code only, such as device drivers.

The restrictions on cmpSize mean that there will be bits left over in each pixel of a 16-bit or 32-bit PixMap. For 16-bit PixMaps,

16 [pixelSize] − (3 [cmpCount] * 5 [cmpSize]) = 1

so 1 bit will be left over. For 32-bit PixMaps,

32 [pixelSize] − (3 [cmpCount] * 8 [cmpSize]) = 8

so 8 bits will be left over. These bits are not used by QuickDraw. In fact, if QuickDraw creates the PixMap, these leftover bits are zeroed out. How-

ever, QuickDraw will not zero out these bits again; you could use these bits for some application-specific purpose.

▶ Support for Direct Color in the PICT2 Format

The PICT2 format is used to store the details of a Color QuickDraw picture, either in memory or on disk. A QuickDraw picture is simply a sequence of QuickDraw commands. Each command is stored as an opcode with its data following.

By the Way ▶ PICT images can be played back on any Macintosh. PICT2 images could only be played back on a Macintosh with a 68020 or later CPU chip because Color QuickDraw was written using some instructions available on a 68020 or later CPU chip. This limitation has been partially removed in 32-bit QuickDraw, as you'll see later in this section.

The original PICT format used 1-byte opcodes. There weren't sufficient opcodes for Color QuickDraw, so the PICT2 format was given 2-byte opcodes. The opcode and its associated data are followed by a pad byte if needed to make the length even. One problem occurs when working with PICT-formatted data with your own routines: The unused opcodes don't have a defined length. If you try to parse a PICT and you run into an undefined opcode, you don't know how long it is. Therefore, even if you want to skip over it, you can't because you don't know where to jump. The latest version of the PICT2 format changes that because the length of every opcode has been defined. Apple has also reserved all unused opcodes for itself. You are not allowed to use a reserved opcode for your own purposes.

Warning ▶ Portions of the PICT format have been defined by Apple in a Technical Note. The other portions of the format, including the description of a QuickDraw region, have not been publicly described because they are an Apple trade secret. This is important because the PICT and PICT2 formats are the basic graphics format used by almost all applications on the Macintosh, and if you ever needed to move images from the Macintosh to another platform, you'd need to know all the details of the PICT format.

The PICT2 format has been extended to support direct color. Two new opcodes support direct color: **DirectBitsRect** and **DirectBitsRgn**. The former opcode is used to store a PixMap (which will always fit into a Rect).

The latter opcode is used to store a region that contains a PixMap. Because the lengths of all opcodes are now defined, the operating system can read a PICT2 image on any machine. The **DirectBitsRect** and **DirectBitsRgn** opcodes are skipped on machines that do not support Color QuickDraw. Therefore, when playing a PICT2 image on a Macintosh SE, you'll see the image less any color or gray-scale PixMaps or regions containing PixMaps. Although this isn't the ultimate solution, it is better than the previous inability to play the PICT at all.

On an unrelated note, the **FontName** opcode has been added to enable developers to specify font names in addition to their IDs. In the past, previous limitations in the operating system led to problems maintaining the relationship between font IDs used on one machine and the IDs used for the same fonts on another machine. Using font names is now the preferred method of saving font information in a document, and this opcode brings the PICT2 format in line with this new method.

By the Way ▶

This problem maintaining the identity of fonts across machines is a major flaw in the original design of the Macintosh operating system. The designers assumed that there would never be more than 512 fonts on the Macintosh. As it turns out, thousands of fonts are now available. The font numbering scheme has been revised once (changing from using FONT resources to NFNT resources). Even that has proven to be insufficient, and now Apple recommends that font names be stored in a document instead of font IDs. Problems occur even with this solution—it turns out that several different fonts with the same name are available from different vendors. Apple recommends that font vendors prefix their company name to the font name. This leads to long, cumbersome font names, but it does (finally!) solve the problem of tracking fonts across machines. TrueType fonts ('sfnt' resources) contain much more font identification information. Refer to Chapter 8 for a description.

One other opcode was added to the PICT2 format: **lineJustify**, which lets you specify the justification of a line of text and the amount of extra space added to the line for justification. This opcode was added to provide support for the Script Manager.

The data following a **HeaderOp** opcode, which is used to store a PICT header, has been redefined. The data now contains the HRes and VRes for the picture, using the largest value occurring in the picture in both cases.

Note ▶ QuickDraw *does* support PixMaps at resolutions other than 72 dpi (the standard screen resolution). The **HeaderOp** opcode, and the new **OpenCPicture** call, make it easier to support PixMaps at these other resolutions.

▶ Using New QuickDraw Calls

Call **OpenCPicture** instead of **OpenPicture** to create an extended PICT2 format file. This call works on all members of the Macintosh family, not just those that support Color QuickDraw. You need to pass the optimal rectangle (for displaying the picture) and the horizontal and vertical resolutions of the picture. When you're done drawing the picture, call **ClosePicture** (as before). To draw it, call **DrawPicture** (also as before).

Call **BitMapToRegion** to convert a bitmap or PixMap into a QuickDraw region. You can then operate on the region using the QuickDraw region calls. You might want to do this to drag the outline of a bitmap using **DragGrayRegion** or to test for mouse hits against the bitmap.

To find out if the drawing operations have all completed, call **QDDone**. You'd use this call if your application was graphics-intensive and if some of your users might be using a graphics accelerator card. These cards operate asynchronously to speed your application.

Last, you can now call **CopyBits** (not a new call) using a new mode: **DitherCopy**. QuickDraw, when copying a PixMap to an indexed device in this mode, tries to minimize the errors in color matching by spreading them around the entire image. Note that the result of doing a clipped **DitherCopy** will not produce the same results as an unclipped **DitherCopy**.

Instead of calling **CopyBits** and then **CopyMask**, you can use the new **CopyDeepMask** routine that combines the two. It copies a BitMap or PixMap using another BitMap or PixMap as a mask. You can use any of the copy modes and, optionally, the new ditherCopy mode.

Call **GetGray**, passing two RGB colors and a handle to a graphics device, to get the optimal color between the two specified colors. You can use this routine to get the best gray for use in graying menu items. This routine is simple to use (and is simpler than writing your own routine) and is especially useful since many people are now using non-monochrome displays.

The **QDError** routine, which was introduced in *Inside Macintosh*, Volume 5, has been enhanced. It no longer fails on machines without Color QuickDraw and can return several new error codes.

▶ Improved Gray-Scale Support

To provide better support for gray-scale images, three default gray-scale color tables now provide evenly spaced sets of grays. Call **GetCTable** and ask for ctID (color table ID) 34, 36, or 40 to get the default gray-scale color table for 2-bit, 4-bit, or 8-bit-wide gray-scale images. Don't forget that this call can also provide the default color tables (which provide a range of colors and *not* grays) by calling **GetCTable** and asking for a ctID of 66, 68, or 72 to get the default color table for 2-bit, 4-bit, or 8-bit-wide color images.

Color QuickDraw will use the default gray-scale color tables when the user has set a display to a gray-scale mode. QuickDraw automatically calculates the luminance of each color in the PixMap to be displayed and chooses the closest gray in the palette.

▶ Telling QuickDraw That You've Been Fiddling

It's never been advisable to alter most data structures that belong to the operating system. Apple has often warned programmers that it is a dangerous habit to get into because it leads to incompatibilities with future versions of the operating system, or with future machines. In fact, this is why many applications on platforms such as MS-DOS have to be modified to run on so-called compatibles. MS-DOS programmers commonly modify operating system data.

You can safely use many routines to modify the fields of a QuickDraw data structure, and you should use these routines if you need to modify a QuickDraw data structure. However, on occasion this rule must be violated—perhaps a routine to fiddle with the data structure you need doesn't exist, or you may be willing to live with the possibility that your application might not run on a future machine or a future version of the operating system.

Warning ▶ Another reason why you shouldn't fiddle with QuickDraw data structures directly is that QuickDraw maintains some private data structures that you don't know about. If you change data structures, the private data structures that QuickDraw maintains might not be in sync with the data structures you're changing.

If you do fiddle directly with QuickDraw data structures, then you'll be pleased to hear that four new routines will let you tell QuickDraw that you've been changing fields in its data structures. These routines (and when they should be called) are as follows.

- **CTabChanged**—After changing the content of a color table
- **PixPatChanged**—After changing the content of a PixPat data structure or either of its substructures (patMap or patData)
- **PortChanged**—After changing the contents of a port or any of its substructures
- **GDeviceChanged**—After changing the contents of a GDevice or any of its substructures

If you have changed the content of the color table (CLUT) referenced by a PixPat, a port, or a GDevice, you also have to call **CTabChanged**.

▶ The Color Picker Package

The Color Picker Package provides a standard dialog box allowing users to select a single color from a 48-bit range using either RGB (Red-Green-Blue) coordinates or HSV (Hue-Saturation-Value) coordinates. The user is presented with a color wheel that allows him or her to quickly go to the approximate range of the desired color.

Warning ▶ If your application requires that users select colors from a smaller range, you'll have to devise your own color selection dialog. There is no way to reduce the color choices in the standard Color Picker dialog.

Call the **GetColor** routine to present the standard Color Picker dialog. You can specify where the dialog box should be located, the prompt for the dialog box, and an initial color. If the monitor is running in color in 4-bit mode or greater, the dialog is displayed in color; in all other cases, the dialog is displayed in black and white. The dialog returns the output color (a 48-bit RGB color) selected by the user if the user has pressed the OK button.

Many different color models are available, and each was derived for some area of the application. The Color Picker supports four color models. RGB is used with video and video monitors, and color is additive: the more red or green or blue, the closer the resultant color gets to white. In the CMYK (Cyan-Magenta-Yellow-blacK) model, used by printers, color is subtractive: that is, the more cyan or magenta or yellow, the closer the color is to black. The HSV (Hue-Saturation-Value) and HLS (Hue-Lightness-Saturation) models, used by graphic artists and designers, are quite similar and cannot be described as simply as the previous two models.

▶ Converting between Color Models

The Color Picker Package provides six routines to convert RGB colors to and from the other three models. These conversion calls are listed in Table 15-1.

Table 15-1. Color model conversion routines

Color Model	*To Convert from RGB to the Model, Call:*	*To Convert from the Model to RGB, Call:*
CMY	RGB2CMY	CMY2RGB
HSL	RGB2HSL	HSL2RGB
HSV	RGB2HSV	HSV2RGB

The RGB model specifies a color by using three 16-bit integers: one for red, one for green, and another for blue. The other three models specify their components as SmallFracts—that is, a number in the range of 0 to 1. Two routines convert to and from SmallFracts: call **SmallFract2Fix** to convert a SmallFract to a fixed integer, or **Fix2SmallFract** to convert a fixed integer to a SmallFract.

▶ The Palette Manager

The Palette Manager manages the color and gray-scale needs of all applications and the operating system across all the monitors on the system. Your application can use the Palette Manager to manage sets of colors or grays, to animate color tables, and to control your application's use of colors and grays.

▶ Types of Colors

You control how colors are used in your application by specifying usage flags for each color in a palette. You can use five flags: courteous, tolerant, animating, explicit, and inhibited.

A *courteous* color accepts the closest match that the Color Manager can find in the CLUT. By using courteous colors, your application will cause no changes in the CLUT. On direct devices, a courteous color produces the specified color (to the limits of the device).

A *tolerant* color either accepts a match from the CLUT within the specified tolerance or adds the exact color to the CLUT. That is, on indexed devices, tolerant colors force their way onto the CLUT if there isn't a color within the range you've specified. On direct devices, a tolerant color produces the specified color (to the limits of the device).

An *animating* color is used to produce animation effects on indexed color devices. Animating colors have no effect on direct devices.

An *explicit* color is produced by the specified index into the CLUT. This is not an especially useful property by itself, but is most often used in combination with either the tolerant or animating property to guarantee that a color gets a specific index in the CLUT.

An *inhibited* color is prevented from appearing on the specified kinds of devices. You can specify the following devices: 2-bit, 4-bit, or 8-bit gray-scale and/or 2-bit, 4-bit, or 8-bit color devices. This property, specifiable in detail, gives you detailed control of your palettes on different types of display devices. For example, Apple recommends that you inhibit tolerant colors when displaying on a gray-scale device because the Palette Manager uses a default CLUT, which in this case produces the best results.

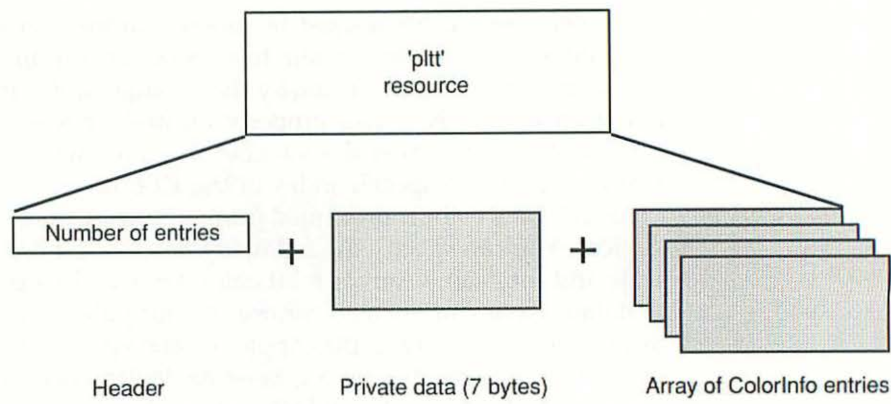
Warning ▶ Test your application on displays set to the various combinations (2-bit, 4-bit, and 8-bit, color or gray-scale) to verify what you're doing with inhibited colors.

▶ Creating and Using 'pltt' (Palette) Resources

You can create a default palette that the Palette Manager will use to color all windows that don't have a defined palette. This is a quick way of getting color onto windows without doing a lot of work. Simply give the 'pltt' resource an ID of 0. If your application has windows without an associated 'pltt' resource and your application doesn't have a 'pltt' resource of ID 0, the Palette Manager will use the default 'pltt' resource in the System (of ID 0). Figure 15-1 shows a 'pltt' resource.

Call **GetNewPalette** to retrieve the specified 'pltt' resource. This routine initializes the private variables in the resource. Don't use the **GetResource** call to retrieve a palette resource, or these variables won't be initialized. Call **NewPalette** to create a new palette with the specified number of entries and color table. You'll also supply a set of usage flags and a tolerance value that will be used for each of the colors in the palette. Call **DisposePalette** to deallocate a palette. Don't use **DisposeResource** to do this, or the Palette Manager will get confused—'pltt' resources contain private data used by the Palette Manager.

To retrieve the specified entry in the specified palette, call **GetEntryColor**. To set the specified entry in a palette, call **SetEntryColor**. Call **GetEntryUsage** and **SetEntryUsage** to get and set the usage and tolerance values for the specified entry in the palette.



Each ColorInfo entry looks like this:

RGB - red component
RGB - green component
RGB - blue component
Color usage flags
tolerance
Private flags and data

Figure 15-1. A 'pltt' (palette) resource

Call **CTab2Palette** to copy the data in a color table to a palette, and call **Palette2CTab** to perform the inverse operation.

▶ Using Palettes with Windows

Call **ActivatePalette** after changing a window's palette (using the Palette Manager routines such as **SetEntryColor** or **SetEntryUsage**). This call generates update events for all windows affected by the changes to the palette if the specified window is frontmost. You should call this routine after making an entire set of changes to a palette, not after each change. The Window Manager also calls **ActivatePalette** whenever your window opens, closes, moves around, or comes to the front.

To get a handle to the palette of the specified window, call **GetPalette**. If you pass the value of −1 for the window pointer to this routine, you'll get a handle to the default palette for your application.

Call **SetPalette** to set the palette of the specified window to the speci-

fied palette. You also specify whether the window should receive update events if a change in its color environment occurs. To set the default application palette, specify a window pointer of −1. If you need more control over exactly when you want your window to receive update events, use **NSetPalette**. This call allows you to specify whether the window should receive update events if there is a change in its color environment when the window is in the frontmost position or not in the frontmost position, always and/or never. Call **SetPaletteUpdates** and **GetPaletteUpdates** to set or get the update attribute of the specified palette.

▶ Working with Palettes

Call **PmForeColor** and **PmBackColor** to set the RGB and index fields of the foreground and background colors, respectively, of the current window to the values of the specified entry in the window's palette. Call **SaveFore** and **SaveBack** to return the current foreground and background colors, respectively. Call **RestoreFore** and **RestoreBack** to set the current foreground and background colors, respectively.

To make a copy of a palette, call **CopyPalette**. Once again, don't use the Resource Manager routines to do this (or your own code) because the Palette Manager maintains its private data in palettes. Call **ResizePalette** to enlarge or reduce the size of a palette. You specify a palette and the number of entries that should be in it.

Call **RestoreDeviceClut** to change the CLUT of the specified device to its default state. If any changes in color happen because of this, update events will be posted to any windows displayed (partially or completely) on this device.

▶ Animating Palettes

Two new system calls have been provided to simplify palette animation. Call **AnimateEntry** to change a single color entry in a palette. Pass a pointer to the appropriate window, an index to the palette entry to be changed, and the new color. No animation will happen if the palette entry is not an animated color.

To simultaneously change a range of colors in a palette, call **AnimatePalette**. Pass a pointer to the appropriate window, an index to the first entry to be changed, a count of the number of entries to be changed, a handle to a new color table, and an index to the first entry in that color table to be copied. This routine then modifies as many entries in the window's color table as possible.

▶ The Graphics Device Manager

The Graphics Device Manager is used primarily by QuickDraw, the Palette Manager, and the Color Manager to communicate with graphics devices, including monitors and output devices. Occasionally, some applications may need to do this, especially applications that are graphics-intensive.

You might find this manager interesting because it also provides routines to create and manage offscreen bitmaps. These capabilities can change users' perceptions of your application, because you can build a picture behind the scene and then quickly display it on the screen. Your application appears more responsive to your users.

The important data structures used by the Graphics Device Manager are the GDevice record, which is used to store all the data needed to describe the device, and the GWorld record, which is used to describe an offscreen bitmap. Let's first look at using offscreen bitmaps and then move onto some of the other things you can do with the Graphics Device Manager.

▶ Creating GWorlds

A GWorld record is an extension of a color GrafPort record (a cGrafPort record). You create an offscreen bitmap as a GWorld. If you're working in color, you need to create an associated GDevice record (the details of GDevice records follow). If you're working in monochrome, you don't need to create a GDevice record.

Also associated with a GWorld record is a set of flags used to control offscreen graphics. These flags, contained in a structure called a GWorld-Flags record, describe the following.

- Whether the GWorld can use temporary memory
- Whether the bitmap is purgeable and/or locked
- Whether the GWorld was modified and in what ways by a call to **UpdateGWorld**

Notice that the GWorld's bitmap can be purgeable. If you allow it to be purgeable, then your code should be capable of handling the situation if the bitmap is purged; in this case, you'll have to recreate the bitmap.

Create a new GWorld record by calling **NewGWorld**. You need to specify the pixel depth, the bounds rectangle, a color table, a set of GWorld-Flags flags, and an optional handle to a GDevice record. The **NewGWorld**

call allocates memory for the GrafPort and the PixMap. Call **OpenCPort** to initialize the GWorld record before using it.

In addition to specifying whether the associated PixMap is purgeable when you create it, you can also control whether the bitmap is purgeable after it's been created by calling **AllowPurgePixels** and **NoPurgePixels**.

Before drawing to the GWorld record, you must lock its PixMap down by calling **LockPixels**. This Boolean call returns, telling you whether the PixMap has been purged. When you're done drawing, call **UnlockPixels**.

Call **GetPixelsState** to find out whether the PixMap has been purged and whether it has been locked. Call **SetPixelsState** to set the state of the PixMap. Use these two calls to save the current state of the PixMap, perform some operation(s) on it, and then restore its previous state.

Use the **UpdateGWorld** routine to change the parameters of a GWorld record, such as its bounds rectangle, pixel depth, and color table. You can specify whether the PixMap should be clipped, stretched, and/or dithered. If the PixMap was purged, this call will reallocate it.

Call **GetGWorld** to get the current port (whether it is a GrafPtr, CGrafPtr, or GWorldPtr) and the current GDevice record. Call **SetGWorld** to set the current port (whether it is a GrafPtr, CGrafPtr, or GWorldPtr) and the current GDevice record.

Call **GetGWorldDevice** to get a handle to the device attached to the specified offscreen world. Call **GetGWorldPixMap** to get a handle to the PixMap record of the specified offscreen world. Use this routine when using the offscreen graphics world routines such as **GetPixelsState**and **LockPixels**.

Before directly accessing a PixMap, call **PixMap32Bit**to find out whether the specified PixMap must be addressed in 32-bit mode. Call **GetPixBaseAddr** to get a 32-bit pointer to the PixMap pixels. To access the pixels, first switch to 32-bit mode, do what you need to do to the pixels, and then switch back to 24-bit mode (if that's where you started). Don't use any QuickDraw routines between your call to **GetPixBaseAddr** and accessing the pixels, or the contents of an offscreen buffer may no longer be accurate.

To deallocate a GWorld record and all its associated memory, call **DisposGWorld**. This call also deallocates the GDevice record if one was created for this offscreen world.

▶ Using GDevice Records

The GDevice record holds all the information needed to control a graphics device. When the system boots up, it creates a linked list of GDevice records for each graphics device that the system finds. You can also create

GDevice records when building an offscreen bitmap. In this case, the record doesn't necessarily describe a physical device—you might create a GDevice record that describes your ideal graphics world. After drawing a picture in this environment, you could copy it to a screen.

The GDevice record contains the following information.

- Device type
- Preferred resolution
- An inverse color table (used to check if a particular color is available in the device's CLUT)
- The boundary rectangle of the GDevice record (relative to the screen with the menu bar on it)
- A handle to the pixel map of the displayed image
- A set of status flags

The record also contains some private data that belong to the Graphics Device Manager.

Several routines enable you to work with GDevice records. Let's look at the routines that might prove useful when developing application software.

To create a new GDevice record, call **NewGDevice** and specify whether it is associated with a physical device. This call does not insert the GDevice record into the linked list maintained by the Graphics Device Manager, but applications shouldn't be doing that anyway. You also specify the mode (monochrome or color) when creating the GDevice record.

To get a handle to the current GDevice record, call **GetGDevice**. This call allocates all the memory required for this record and the handles in the record. When it finishes, it returns a handle to the current GDevice record. To get a handle to the main GDevice—that is, the GDevice record for the screen with the menu bar—call **GetMainDevice**. To get a handle to the GDevice with the deepest device—that is, the GDevice record with the most bits per pixel—call **GetMaxDevice**.

Call **GetDeviceList** to get a handle to the first device in the global device list. You can then call **GetNextDevice**to get the next device in the list. When there are no more devices in the list, this call will return NIL.

Call **DeviceLoop**, passing the address of a drawing procedure and a handle to your drawing region, to search all active devices and call your drawing procedure whenever a device intersects your drawing region. You also can pass 4 bytes of (user) data and some control flags.

Call **TestDeviceAttribute** with a handle to a GDevice record to check

for a particular attribute, such as whether the device supports color or if it is a screen device. You can call **SetDeviceAttribute** to set attributes for a device, but few applications will need to use this call.

To check whether a device has a particular pixel depth and for its attributes, call **HasDepth** with a handle to a GDevice record. You can call **SetDepth** to set the pixel depth and attributes, but set the pixel depth only if the user has given you permission to do this. Do not change the pixel depth of the monitor without asking the user first.

When you're about to draw to an offscreen bitmap, call **SetGDevice** to make it the current device. You shouldn't need this call for any other purpose.

Last, call **DisposGDevice** to deallocate a GDevice record and its associated handles. You'll only need this call when disposing of offscreen bitmaps.

▶ The Picture Utilities Package

The Picture Utilities Package contains routines that let you examine PICT files and PixMap records. By examining a PICT file with these routines, you can know more about its contents, such as which fonts and colors were used, the horizontal and vertical resolution, and the maximum pixel depth. This package is new starting with System 7.

▶ Examining PICT Files

By calling **GetPictInfo** and passing it a handle to the PICT file and a PictInfoRec data structure, the routine will walk through the file and store information about the PICT into the PictInfoRec. The following information is returned:

- Horizontal and vertical resolution of the picture
- The rectangle that contains the picture at its default resolution
- The number of lines, rectangles, rounded rectangles, ovals, arcs, polygons, regions, PixMaps, comments, and text occurring in the picture
- The number of different comments in the picture
- The number of comments of each type (optional)
- The number of different fonts in the picture
- The size(s) and style(s) used for each font (optional)

- The name(s) of the font(s) (optional)
- A PixMapInfoRec that describes the contents of the PixMap record(s) in the picture (PixMapInfoRec records are described below)

If you need to collect this information about more than one PICT, there's another set of routines to make this easier. First, call **NewPictInfo** to get a unique ID for this set of queries. This call also sets up some private storage to collect the PICT information. Then call **RecordPictInfo**, passing it the unique ID and a handle to the PICT for each PICT you want to examine.

When you want to examine the results, whether or not you're done running PICTs through the **RecordPictInfo** routine, call **RetrievePictInfo**. This routine returns a PictInfoRec data structure, similar to **GetPictInfo**, but the horizontal and vertical resolutions in the record are the highest resolution encountered. When you're done examining PICTs, call **DisposePictInfo** to allow the Graphics Utility Package to dispose of its private storage.

▶ Examining Pixel Maps

By calling **GetPixMapInfo** and passing it a handle to the PixMap record and a PixMapInfoRec data structure, the routine will examine the PixMap and store information about it into the PixMapInfoRec record. The following information is returned:

- Horizontal and vertical resolution of the PixMap
- The pixel depth
- The number of colors occurring in the PixMap
- A handle to a palette containing all the colors in the PixMap (optional)
- A handle to a color table containing all the colors in the PixMap (optional)

You specify the maximum number of colors you'd like (in the palette and/or color table). You also specify whether the colors selected should be the widest range or the most popular.

If you need to collect this information about more than one PixMap, there's another set of routines to make this easier. First, call **NewPictInfo** to get a unique ID for this set of queries. This call also sets up some private storage to collect the PixMap information. Then call **RecordPixMapInfo**,

passing it the unique ID and a handle to the PixMap for each PixMap you want to examine.

When you want to examine the results, whether or not you're done running PixMaps through the **RecordPixMapInfo** routine, call **RetrievePixMapInfo**. This routine returns a PixMapInfoRec record, similar to **GetPixMapInfo**, but the horizontal and vertical resolutions in the record are the highest resolution encountered. The depth is the deepest encountered, and the color table and/or palette contain colors from all the PixMaps. When you're done examining PixMaps, call **DisposePictInfo** to allow the Graphics Utility Package to dispose of its private storage.

▶ Customizing the Picture Utilities

One of the parameters that you pass when calling the **GetPixMapInfo** and **GetPictInfo** routines tells them how to sample color. You can ask that colors be sampled so that you get either the most popular colors or the widest range of colors. Unless you specifically require one of these sampling methods, you should let the utilities choose the sampling method. By doing this, your application will be compatible with future versions of the Picture Utilities, which may support other sampling methods.

You can also provide your own color sampling method. You'll need to write four routines to do this:

- **InitPickMethod**—allocates any storage required and initializes method
- **RecordColors**—records colors if you create your own custom color bank (the default is to store colors in a 5-5-5 (RGB) bit deep histogram)
- **CalcColorTable**—calculates how many unique colors were added to the color bank and returns the requested number of colors from the color bank
- **DisposeColorPickMethod**—deallocates any memory allocated in the InitPickMethod routine

▶ Conclusion

In this chapter, you've seen all the changes to QuickDraw. Most of these changes were previously available under System 6 as a separate package known as 32-bit QuickDraw. This package has been integrated into the operating system with System 7. The biggest change is that direct color

devices are now supported. Color images can be displayed on such devices with photographic realism.

Get Info ▶

For more information on Color QuickDraw, refer to the Color QuickDraw chapter and the Graphics Overview chapter in *Inside Macintosh*, Volume VI. You should also review the Color Quick-Draw chapter in *Inside Macintosh*, Volume V, and you might also want to review the QuickDraw chapter in *Inside Macintosh*, Volume I.

For more information on the Color Picker Package, refer to the Color Picker Package chapter in *Inside Macintosh*, Volume VI. Note that this chapter completely replaces the Color Picker Package chapter in *Inside Macintosh*, Volume V.

For more information on the Palette Manager, refer to the Palette Manager chapter in *Inside Macintosh*, Volume VI. Note that this chapter completely replaces the Palette Manager chapter in *Inside Macintosh*, Volume V.

For more information on the Picture Utilities Package, refer to the Picture Utilities Package chapter in *Inside Macintosh*, Volume VI.

For more information on the Graphics Device Manager, refer to the Graphics Device Manager chapter in *Inside Macintosh*, Volume VI. Note that this chapter completely replaces the Graphics Device Manager chapter in *Inside Macintosh*, Volume V.

16 ▶ The Memory Manager

▶ Introduction

This chapter covers the two large changes in the Memory Manager brought by System 7. The first and most obvious change is virtual memory, which allows users to use disk space to simulate more expensive RAM. The addition of virtual memory affects few applications directly, as you will see, but it will make life easier for many users.

The second change is the generalization of the temporary memory calls, which were introduced with MultiFinder. The restrictions on using temporary memory meant that few applications took advantage of it. With System 7, most of these restrictions have been removed or at least loosened.

Last, you'll look at compatibility with 32-bit addresses (known as 32-bit cleanliness), which is important for compatibility with System 7. System 7 can operate in 32-bit mode now, and applications that are 32-bit clean should indicate this in their 'SIZE' resource. The Finder will warn users to beware if they launch an application that does not indicate this. Such an application may very well run in 32-bit mode, but the warning message will make users a bit nervous. This is a good time to clean house and set that bit.

▶ Virtual Memory

Virtual memory (or VM) allows users to simulate RAM on a hard disk. A user might have only 2 Mb (megabytes) of physical RAM, but by using disk space, he or she could run various applications as if there were, for

example, 4 Mb of RAM. This is done by mapping logical memory onto a combination of physical memory and disk sectors. The operating system keeps track of memory in pieces known as *pages*. Under System 7, pages are 4K long.

When a page of memory is needed but is not in physical RAM (this occurrence is known as a *page fault*), the operating system goes to the disk, retrieves the page, copies it to memory, and allows the application to proceed. This process must happen so quickly that it is implemented in hardware. The hardware requirement is that an MMU, or Memory Management Unit, must be available. Any Macintosh that uses a Motorola 68030 or later CPU chip has an MMU built into the CPU. Any Macintosh that uses a Motorola 68020 must have a separate 68851 MMU. Macintosh computers that use a 68000 cannot have virtual memory because there is no MMU chip designed for use in these machines.

The Memory Manager creates a *backing store* file to hold memory pages on a disk selected by the user (called the *paging device*). The backing store file must be as large as the total size of virtual memory, but it does not have to be contiguous, allowing for a simple, one-to-one mapping between logical memory and the backing store file. In the example, if the user wants 5 Mb of virtual memory, then the Memory Manager will create a 5 Mb file on the paging device.

The paging device must be an HFS volume, because currently the store must be capable of block-level I/O calls. AppleShare volumes and foreign file systems (A/UX, Apple II, and MS-DOS, for example) cannot do this under the Macintosh operating system today.

By the Way ▶

> The backing store file does not have to be contiguous. If it was required to be contiguous, many users would have to back up their disk, reformat it, and restore their files to use virtual memory. This is because files tend to become fragmented on a disk over time. However, a noncontiguous backing store file can be created at any time, as long as sufficient disk space is available on the disk.

The architecture of the virtual memory system is controlled primarily by the basic design of the Motorola MMUs and CPUs. Refer to the manuals for these chips for more information on the low-level details of implementing VM. The form of VM implemented under System 7 is known as *demand paging*—pages of VM are brought into RAM when they are required (on demand).

Few applications need to work directly with virtual memory. Those that do need to work directly with VM will have critical timing requirements.

Device drivers may need some virtual memory system calls for timing reasons as well, or because they are executing at interrupt time. This topic is discussed below.

Applications should not use privileged instructions, because alternate ways of accomplishing the same purposes (such as checking for a math coprocessor) are available. Until System 7, the operating system and applications always ran in Supervisor mode on the 680x0 CPU. VM runs in Supervisor mode and applications now run in User mode. To maintain compatibility with the small number of applications that did use Privileged instructions, the operating system emulates some of these instructions in software. This does affect performance.

Debuggers also need to know about virtual memory. System 7 provides a set of calls for debuggers to make their way around virtual memory. Refer to the Memory Manager chapter in *Inside Macintosh,* Volume VI for more details on these routines.

A schematic of 24-bit virtual memory is illustrated in Figure 16-1. Virtual memory in a 24-bit world is confined to 16 Mb. The 255 other 16 Mb slots are not used except to map NuBus slots, ROM, and I/O into 32-bit mode. Of the 16 slots having 1 Mb, slot 8 is used for the system ROM and slot 16 for I/O space. The maximum amount of virtual memory in a 24-bit machine is therefore 14 Mb. The memory slots between these two slots can be used either by a NuBus card or for virtual memory. Last, each 1 Mb slot is mapped into 256 pages, each of a length of 4K.

The virtual memory system implemented in System 7 does not provide memory protection. One application can still write to memory used by another application, potentially causing it to crash or to corrupt its data. This version of virtual memory also does not provide multiple address spaces, so MultiFinder memory can still be fragmented, reducing the performance of the machine.

▶ Compatibility and Virtual Memory

Check that an MMU and virtual memory exist before making any virtual memory-related calls to the Memory Manager. You can do this by first calling **Gestalt** with a selector of gestaltMMUType. This call returns with a value indicating no MMU is present, an AMU (Address Management Unit in Macintosh IIs) is present (this isn't an MMU), a Motorola 68851 PMMU is present, or a Motorola 68030 with built-in PMMU is present.

Next, call **Gestalt** with a selector of gestaltVMAttr. This call returns with a description of the virtual memory system. If you're curious about the virtual memory page size, call **Gestalt** with a selector of gestalt-LogicalPageSize. To find the amount of logical and physical memory, call

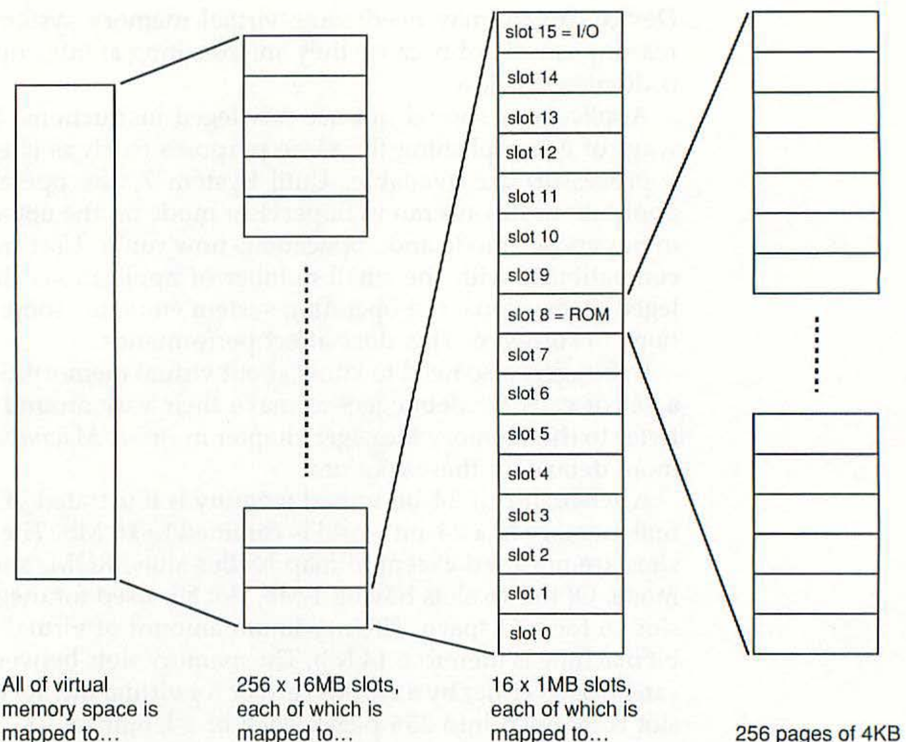


Figure 16-1. Schematic diagram of 24-bit virtual memory

Gestalt with selectors of gestaltLogicalRAMSize and gestaltPhysical-RAMSize.

▶ Controlling Virtual Memory

A page or range of pages of virtual memory can be *held* in physical memory. This means that the page will always be in physical memory, but it can move around. A page or range of pages can be **locked** into physical memory. This means that the page will always be in physical memory and that it cannot move around. For performance reasons, try to hold pages of VM rather than locking them. Holding is faster and requires less overhead than locking does.

Whenever a page or range of pages is held or locked, they must be unheld or unlocked. If a page is held twice, it must be unheld twice. This also holds true for locking pages.

The **HoldMemory** call forces the Memory Manager to hold the pages containing the given range of memory. This call rounds down to the near-

est page boundary and up to the nearest page boundary so that all the given addresses will be on held pages. The **UnholdMemory** call marks the pages containing the given addresses as unheld.

The **LockMemory** call forces the Memory Manager to lock the pages containing the given range of memory. This call rounds down to the nearest page boundary and up to the nearest page boundary so that all the given addresses will be on locked pages. The **LockMemoryContiguous** call locks the pages into contiguous physical memory. The **UnlockMemory** call marks the pages containing the given addresses as unlocked.

The **GetPhysical** call takes one or more logical addresses and translates them into physical addresses. This call is used by drivers that need to know the physical addresses corresponding to a logical address.

When the operating system is handling a page fault, the only other code that can execute at that time is at interrupt level. If that code causes a page fault, the result is known as a *double page fault*. The operating system does a lot of things to prevent double page faults from happening. For example, Time Manager tasks, I/O completion routines, and VBL (Vertical Blanking) tasks are all deferred until any paging operations have been completed. The operating system maintains these tasks in a queue, which guarantees that they will be executed in the same order in which they would have been.

The **DeferUserFn** function allows interrupt handlers to defer code that may cause page faults until a later (and safer) time. It might be used, for example, in an I/O completion routine. The function checks whether the call to the completion can be made safely. If it can, it is executed. If not, the routine address and its parameters are stored, and the routine is executed when it is safe to cause a page fault.

▶ So Who Does Have to Worry about Virtual Memory?

Memory-intensive applications, such as gray-scale and color paint programs, scanner software, and multimedia applications, will not be able to use virtual memory to store large images, because virtually all mass storage devices are on the SCSI bus. The paging device is also on the SCSI bus. If you try to read in a large image (or many smaller images) into virtual memory, the images will come from a file on disk into memory, only to be spooled back out to the backing store file on disk. This stalemate will slow the application's performance enormously. Memory-intensive applications should warn users if there is not enough RAM to run efficiently.

Most applications do not have to do anything to take advantage of VM. One of Apple's goals in implementing VM is that it should be transparent to most applications. For the most part, Apple has succeeded in doing this.

One concept that does affect the relationship between applications and VM is known as *locality of reference*. If your routines tend to call other

routines and reference data that are located nearby, then your application is said to have locality of reference. Such applications will tend to perform better under VM than other applications because VM will tend to have nearby code and data in memory already.

You need to write device drivers with a little more care if they are to run under System 7. Load drivers into the System heap, not an Application heap. Use the **Read** and **Write** calls (part of the Device Manager) to transfer data, because the operating system will automatically ensure that the buffers used in these calls stay in RAM. If your application talks to hardware directly, move the code that does talk to hardware into a driver. Although your application might run under System 7 in its current form, by moving hardware-dependent code into a driver today, you'll have fewer problems in the future.

NuBus drivers are affected by virtual memory the most. Actually, only drivers for cards that can be NuBus masters will have to worry about virtual memory. Passive cards (such as video cards) should have no problems under VM, because these cards can control the NuBus and read and write to system memory. Under virtual memory, they will have to ensure that when reading from or writing to memory, they are using a valid physical address, since cards do not address memory through the MMU. This is the same issue as with Macintosh IIci memory cache cards.

▶ Temporary Memory

When temporary memory was introduced with MultiFinder, it was accessed through a special set of system calls and allowed applications to use memory that was not allocated to applications for short periods of time. Applications were expected to return this memory before the next call to **GetNextEvent** or **WaitNextEvent**. The system required temporary memory calls to access these chunks of memory—you could not use regular Memory Manager calls on temporary memory. These restrictions discouraged most developers from ever using these calls. System 7 has loosened or eliminated most of the restrictions.

Why use temporary memory in any case? Temporary memory is a potential source of memory for input/output buffers (the Finder uses temporary memory for copying files), for storing the results of idle-time calculations, and so on. One complication that arises from using temporary memory is handling the case when there isn't enough (or any) available. This does complicate your code, but using temporary memory can enable your application to run more quickly. Using temporary memory also means that the recommended memory partition size can be smaller in some cases.

Also, most applications can use extra memory during a handful of operations, such as opening, closing, or printing documents. Since these are a

small subset of all the functions that applications typically do, you may find it relatively easy to use temporary memory in these circumstances.

▶ Differences in Temporary Memory: Now and Then

Under System 6, temporary memory had to be released before the next call to **GetNextEvent** or **WaitNextEvent**. The short amount of time that temporary memory was available undoubtedly restricted its potential uses. Under System 7, temporary memory can be kept for an arbitrarily long time, although good citizens will return the memory as soon as possible.

Another restriction in using temporary memory under System 6 was that a specialized set of calls had to be used. You couldn't use **Dispos-Handle** on a temporary memory block; you had to use **MFTempDispos-Handle** (now called **TempDisposHandle**), which required temporary memory blocks to be handled with special code. Under System 7, you still have to allocate temporary memory through special calls (that's reasonable), but you can use most of the regular Memory Manager calls to work with temporary memory blocks.

If your application does hold onto temporary memory, the user may not be able to start another application. This condition might continue until either your application lets go of the temporary memory or the user quits an application.

The operating system tracks each application's use of temporary memory. When your application quits (or crashes), any temporary memory it's still using is returned to the system. Therefore, if your application uses VBL or Time Manager tasks that live on beyond your application, do not pass them references to temporary memory.

▶ Compatibility and Temporary Memory

Before you use any of the temporary memory calls, check to see if they are available. To do this, call **Gestalt** with a selector of gestaltOSAttr. If the results show that the gestaltTempMemSupport bit is on, then temporary memory routines are available. If the gestaltRealTempMemory bit is on, then you can use the normal Memory Manager routines to operate on temporary memory blocks. Last, if the gestaltTempMemTracked bit is on, then you can use temporary memory blocks for an arbitrary time.

▶ Temporary Memory Calls

With all these changes, the only temporary memory calls you might need to use under System 7 are **TempFreeMem**, **TempMaxMem**, and **Temp-NewHandle**. Three of the old calls are now obsolete, although they still

work: **TempDisposHandle**, **TempHLock**, and **TempHUnlock**. The **TopMem** call is obsolete as well—The original purpose of this call was to return the total amount of usable memory, and this is now better determined by a call to **Gestalt**.

Note ▶ The temporary memory routines (introduced under System 6) **MFFreeMem**, **MFMaxMem**, **MFTempDisposHandle**, **MFTempHLock**, **MFTempHUnlock**, **MFTempNewHandle** and **MFTopMem** have been renamed under System 7 primarily by removing the **MF-** prefix. The old routine names will still work under System 7 for compatibility.

You can use the following Memory Manager routines with temporary memory blocks:

- **DisposHandle**
- **EmptyHandle**
- **GetHandleSize**, **SetHandleSize**
- **RecoverHandle**
- **ReallocHandle**
- **HandleZone**
- **HLock**, **HUnlock**
- **HPurge**, **HNoPurge**
- **HSetRBit**, **HClrRBit**
- **HGetState**, **HSetState**

You might prefer to check how much temporary memory is available before trying to allocate it. The preferable way to do this is to call **TempMaxMem**, which first compacts the MultiFinder heap zone and returns the size of the largest contiguous free block. **TempFreeMem** returns the total amount of free memory. You may not be able to allocate a block that large, because it might not be contiguous.

You should always check whether the handle returned by **TempNewHandle** is empty. If it is empty, check the error code. Memory blocks returned by **TempNewHandle** will be unlocked and purgeable. If you do use **HandleZone** to figure out where the temporary memory handle came from, don't try to make new blocks in that zone or perform heap operations. You shouldn't care where this memory comes from.

▶ 32-Bit Cleanliness

Macintosh computers with 32-bit clean ROMs (the IIci, IIsi, LC, and the IIfx are examples) can be run in either 32-bit mode or 24-bit mode, at the user's discretion. All other Macintosh computers can be run only in 24-bit mode.

Thirty-two bit cleanliness has nothing to do with the standard size of integers that your compiler uses; it has to do with the way you use memory addresses. If your code makes no assumptions about the structure of pointers, handles, and all other Memory Manager data structures, then your code is probably 32-bit clean and can run in 32-bit mode. Figure 16-2 shows a 24-bit memory map; Figure 16-3 shows a 32-bit memory map.

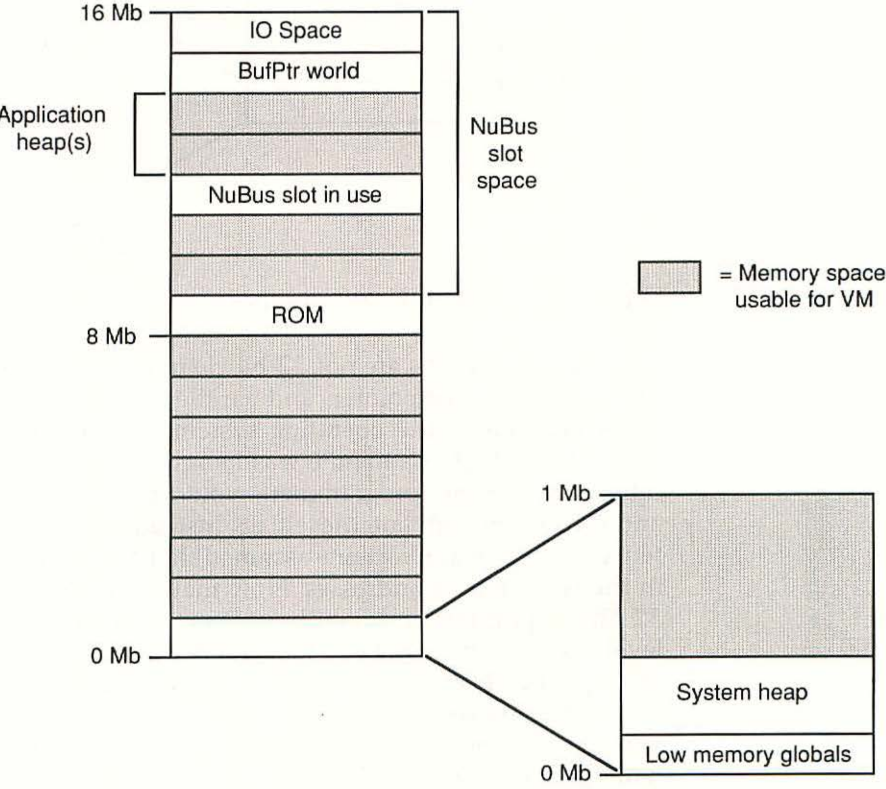


Figure 16-2. A 24-bit memory map

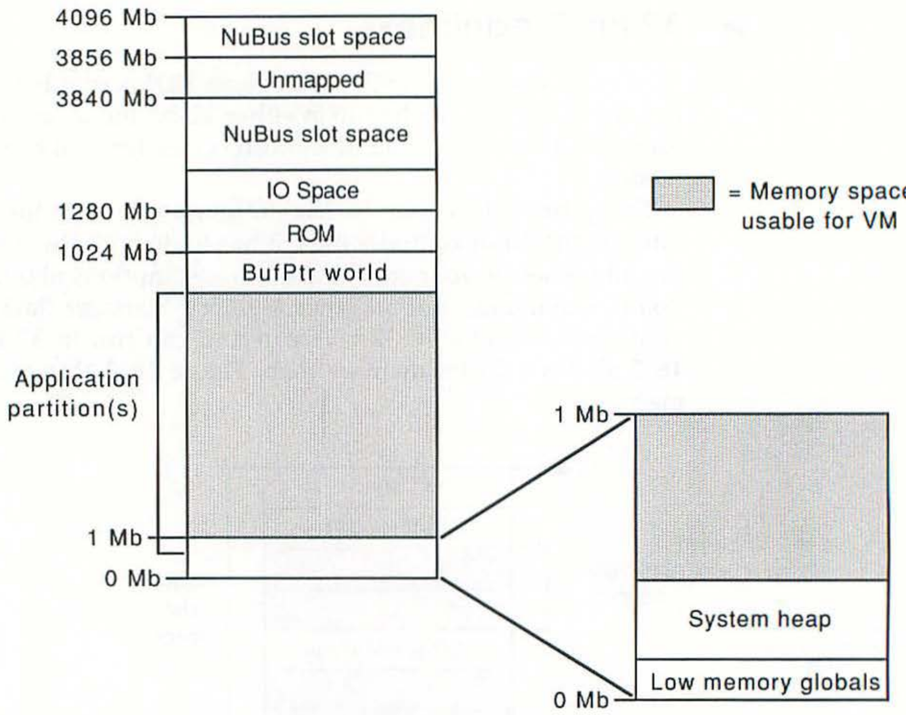


Figure 16-3. A 32-bit memory map

Some applications are not 32-bit clean because they were written with the knowledge that pointers and handles were 24-bit numbers under previous versions of the operating system. The Memory Manager kept its status bits (whether a handle was locked, purgeable, or a resource) in the other 8 bits. Some programmers would try to save a few microseconds and directly manipulate these flags instead of using the appropriate system calls. Other applications assumed that additional data could be stored in the last 8 bits of addresses. Now that the operating system can use all 32 bits of pointers and handles, it's pretty obvious that altering system-level data structures is a bad idea. It always was—the Tech Support group at Apple has been warning about this sort of thing for years.

A/UX also uses all 32 bits of addresses, and there's no reason why the vast majority of Macintosh applications cannot run under it. Before System 7 was released, A/UX provided a test bed to check whether an application was 32-bit clean.

In several cases, you need to be especially careful to remain 32-bit clean: when writing WDEFs (window definition procedures) and CDEFs

(control definition procedures), and when using memory addresses in calculations.

▶ WDEFs, CDEFs, and Cleanliness

Under previous versions of the operating system, both the Window Manager and Control Manager kept the variant code in the last 8 bits of the handle to the definition procedure. Under System 7, you can avoid this by using two new system calls, **GetWVariant** and **GetCVariant**, to retrieve the variant code. These calls are available only under System 7, so if the definition procedure can operate under both Systems 6 and 7, your code will have to handle both cases.

CDEFs have one more problem that needs to be dealt with: the calcCRgns message used the high-order bit as a flag. This was the only way to write a CDEF, so there was no way around this problem until System 7. The Control Manager under System 7 provides two new messages, calcCntlRgn and calcThumbRgn, which replace the previous message and flag. Once again, this is only available under System 7, so if the definition procedure can operate under both Systems 6 and 7, your code will have to handle both cases.

▶ Calculations on Memory Addresses

The **StripAddress** function, a routine which predates System 7, was provided as part of the Memory Manager to provide a safe way to get only the address portion of a handle. This is needed when running in 24-bit mode under System 7 because only the first 24 bits are a memory address. NuBus device drivers needed to call **StripAddress** because the driver might briefly switch the CPU into 32-bit mode to access a hardware address on a NuBus card. At these times, whenever the driver was using the address of a heap object, it would need the address of that object without anything else.

You also had to use **StripAddress** when performing arithmetic on memory addresses, such as comparing two addresses. If any of the last 8 bits were set by the Memory Manager, then the comparison could be invalid.

When the Memory Manager is operating in 32-bit mode, the **StripAddress** function does nothing. A/UX (and future versions of the Macintosh operating system) will always run in 32-bit mode. At that time, the **StripAddress** function will be obsolete.

Call **Translate24To32** to translate a 24-bit address into a 32-bit address

space. When running in 24-bit mode, this routine may not return an address that can be used.

▶ Passing Along the A5 World

Various pieces of code are not part of an application, yet need to address the application globals. Examples of this code include Time Manager tasks, VBL tasks, and notification requests. Two routines, **SetCurrentA5** and **SetA5**, can be helpful in getting your application's current A5 value and setting the A5 register, respectively.

By the Way ▶ The term *A5 world* refers to the application globals, QuickDraw globals, and the application jump table. All of them are accessible through the A5 register of the CPU. The operating system sets up and maintains all three sets of data.

▶ Compatibility and 32-Bit Cleanliness

Applications that are 32-bit clean should indicate this by setting the is32BitCompatible flag in their 'SIZE' resource. If this flag is not set, then the Finder will warn the user that the application may not be safe and that they might consider rebooting into 24-bit mode. This will happen even if your application really is 32-bit clean and that bit is not set.

▶ Conclusion

In this chapter, you've looked at two changes in the Memory Manager. Virtual Memory allows the user to simulate more expensive RAM with less expensive disk space. Users need enough RAM for their average usage rather than RAM for their maximal usage of memory.

Temporary memory, which was available only under rather restrictive conditions, is now more readily available. Applications that occasionally require larger amounts of memory can take advantage of temporary memory. Rather than choosing slower performance (living with smaller amounts of memory) or lots of memory (even though some of this memory is only occasionally needed), you now have a third choice.

You've also looked briefly at the concept of 32-bit cleanliness. If your application is not 32-bit clean, your users will get annoying warning messages when running under System 7, which will not please them. Your application may also fail to run, which will really annoy your users.

Get Info ▶

For more information on virtual memory (VM) and temporary memory, refer to the Memory Manager chapter of *Inside Macintosh,* Volume VI. You may also want to refer to the Memory Manager chapters in *Inside Macintosh,* Volumes I and IV. For more information on the chips in the Macintosh that form the basis of VM, refer to the *MC68851 Paged Memory Management Unit User's Manual* and the *MC68030 Enhanced 32-Bit Microprocessor User's Manual,* both from Motorola Corporation. Phil Goldman wrote an excellent article on Macintosh Virtual Memory in the November 1989 issue of *Byte.* His article clearly explains the inner workings of the VM system.

17 ▶ Processes

▶ Introduction

In this chapter, you will look at three managers that are associated with processes on the Macintosh: the Process Manager, the Notification Manager, and the Time Manager. The Process Manager is a new manager provided by System 7. The other two have been enhanced with the release of System 7.

The first manager discussed in this chapter is the Process Manager. The Macintosh operating system does not yet support preemptive multitasking, even with System 7. In the meantime, in the MultiFinder model of cooperative multitasking, each application depends on all other running applications to behave in a reasonable and cooperative manner. This means you can't hog all available memory and so on. The Process Manager, introduced with System 7, lays some of the groundwork required for preemptive multitasking.

System 7 brings MultiFinder completely into the operating system. It's now so much a part of the operating system that users can no longer turn it off—there is no longer a mode that runs one, and only one, application.

You'll then move on to the Notification Manager, which provides a way for applications running in the background to communicate in reasonable ways with the user. This manager was introduced with MultiFinder and has since been revised.

Last, you'll look at the Time Manager, which provides scheduling services for setting routines to run at a certain time or at periodic intervals. This manager has also been revised for System 7.

The Process Manager

The Process Manager provides services to launch applications, launch desk accessories, and get information about your application and other processes. System 7 introduces a new concept: process serial numbers. You'll look at this concept first and then at how processes are scheduled. Then you'll look at how to use the new routines provided by the Process Manager.

Process Serial Numbers

The Process Manager assigns each active application or desk accessory a unique *process serial number* (PSN). A process serial number is 64 bits long, and the meaning of the bits in a PSN is private to the Process Manager. Process serial numbers are used in specifying addresses for high-level events or in controlling other processes.

The concept of process serial numbers requires the concept of a process. Under System 7, a *process* is an application that has been launched and has an A5 world. An application in this context includes applications (files of type 'APPL') and desk accessories (files of type 'DFIL').

Process Scheduling and Switching

Once your application has been scheduled to run, it doesn't relinquish control until it calls the Event Manager. Specifically, your application can be switched only when it calls **WaitNextEvent**, **GetNextEvent**, or **EventAvail**. Another application cannot gain control until your application calls one of these three routines, whether your application is running in the foreground or background.

With *cooperative multitasking*, your application calls one of these routines regularly whether you're running in the foreground or background. Otherwise, processes running in the background or foreground, respectively, won't get enough CPU time. Low-level code, such as I/O completion routines and device drivers, will still get some CPU time in any case.

You can switch your application in two ways: a major switch or a minor switch. A *major switch* happens when your application is switched from the foreground to the background (or vice versa). A *minor switch* happens when your application is running in the foreground and, because you have no events to process, processes in the background are given some CPU time. Note that your application won't do anything while in the background unless you've coded it to do something there.

A user causes your application to make a major switch by bringing another application into the foreground (if you're running in the fore-

ground at this time) or by bringing your application into the foreground (if you're running in the background at this time).

Let's look at the first case in a little more detail. The second case is identical, except that the names of the applications are different. Suppose your application, MacWidget, is running in the foreground and cannot do anything while in the background. The user clicks on a window belonging to another application, MacMyGarden, to design a garden. Let's also assume that both applications have been written to work with System 7, and they specifically understand how to work with suspend and resume events. The following things happen to make this context switch happen.

- The user has been using MacWidget.
- The user clicks on a window belonging to MacMyGarden.
- The Process Manager is told to bring about the context change, so it issues a suspend event to MacWidget.
- The next time MacWidget calls **WaitNextEvent**, it gets the suspend event.
- MacWidget prepares to switch into the background.
- MacWidget calls **WaitNextEvent** and is suspended.
- The Process Manager saves the context of MacWidget.
- The Process Manager restores the context of MacMyGarden.
- The Process Manager sends a resume event to MacMyGarden.
- The Process Manager resumes the execution of MacMyGarden.
- MacMyGarden, which was suspended on a call to **WaitNextEvent**, gets the resume event.
- MacMyGarden prepares to run in the foreground.
- MacMyGarden calls **WaitNextEvent** and waits for further input from the user.

Minor switches are simpler in nature. Figure 17-1 illustrates what happens during a minor switch. If your application calls **WaitNextEvent** and there are no events for your application, then your context is saved and the context of the next scheduled background application is restored. The background application then gets CPU time but is not moved to the foreground. The only kind of event that an application running in the background can get is a null event, an update event, or a high-level event. The background application runs until it calls **WaitNextEvent** again. Its context is then saved, and the foreground's context is restored and set running again.

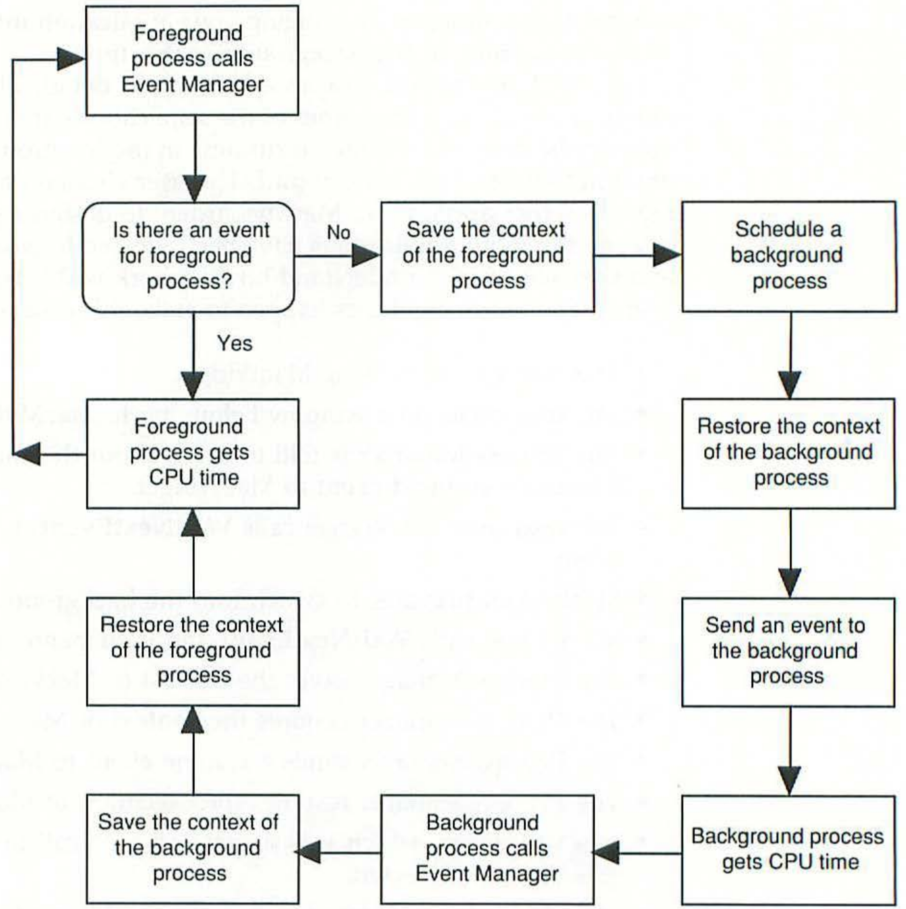


Figure 17-1. What happens during a minor switch

▶ Launching Applications

Call **LaunchApplication** to launch the specified application. You must also supply whether your application should be terminated after the launch with the launchContinue flag. If your application does not terminate, the other application is launched the next time you call an Event Manager routine such as **WaitNextEvent**. You can also specify various control flags, in addition to the launchContinue flag, as follows.

- launchUseMinimum—To launch the application in the smallest partition possible (that is, greater than or equal to the minimum partition size, and hopefully less than the maximum partition size)

- launchDontSwitch—To launch the application into the background rather than the foreground
- launchAllow24Bit—To launch an application that does not have the Is32BitCompatible bit set in its 'SIZE' resource
- launchInhibitDaemon—to disallow launching a background-only application

If you want to be notified when the process you're launching terminates, set the acceptAppDied flag in its 'SIZE' resource. Your application will receive an "unexpected Quit" AppleEvent when the other process terminates.

You can specify an optional high-level event that will be sent to the application as soon as it is launched. For more information on high-level events, see Chapter 7 in this book.

The **LaunchApplication** routine returns the process serial number, the preferred partition size, the minimum partition size, and the maximum available partition size. The last value is returned only if the application couldn't be launched because of a memFullErr error.

When the Finder launches an AppleEvent-aware application, it does not use the application parameters used in earlier systems. Instead, it launches the application and passes the application parameters using AppleEvents.

The **LaunchApplication** routine replaces the old **Launch** routine. This old routine still works for compatibility, but new applications should use the new call.

▶ Launching Desk Accessories

Call **LaunchDeskAccessory** to launch the specified desk accessory. (However, you should use the **OpenDeskAcc** call to launch a desk accessory when the user chooses it from the Apple menu.) The desk accessory is given its own partition and launched in the system heap unless the user holds down the Option key. In that case, if the desk accessory is in an open resource fork available to the launching application, the desk accessory is opened in the application's heap. If the desk accessory is already open, it is brought to the foreground.

▶ Getting Information about Other Processes

Call **GetCurrentProcess** to get the process serial number of the current process. The current process is the process associated with the currentA5 low-memory global. This call works whether the current application is in

the foreground or background. Your application can use this call to find out its process serial number. A driver or other low-level code can use this call to find out the current process. You might use **GetCurrentProcess** to locate an address to send a high-level event.

You can get a list of all current processes by calling **GetNextProcess**, passing it a process serial number, until you get an error of procNot-Found. The first time you make this call, specify a process serial number of kNoProcess, a predefined value. This will return the first process serial number. The next time you make this call, specify the first process serial number and you'll get the second process serial number, and so on.

Call **GetProcessInformation** to get the scoop on the process with the specified process serial number. This call returns the following information:

- Whether the process is an application or desk accessory
- The name of the process as it appears in the application menu
- Creator type and signature
- The address and size of its memory partition
- The application that launched it and the time when it was launched
- The location of the application or desk accessory file
- Contents of the application's 'SIZE' resource

You can use this call on the current application by using the constant kCurrentProcess as the process serial number.

To get the process serial number of the process running in the fore-ground, call **GetFrontProcess**. If you've developed an application as two pieces—a front-end application that handles the user interface and a back-end application that handles computations—the back-end applica-tion could use this call to find out if the front-end application is in the foreground or not. The back-end application might wait to send messages until the front-end application is in the foreground.

Call **SetFrontProcess** to bring the process specified by a process serial number to the foreground. This process will be brought to the foreground when the current foreground process makes its next call to the Event Manager.

Use **WakeUpProcess** to reschedule a process suspended by its last call to **WaitNextEvent**. This call will not move the specified process higher in the queue for execution, but it will be scheduled for execution as soon as its turn arrives.

To compare two process serial numbers, call **SameProcess**. This is the only way that you should compare process serial numbers, because some bits are private to the Process Manager in these numbers and will confuse a simple arithmetic comparison.

▶ Compatibility and the Process Manager

Before using the Process Manager, verify that it is available by calling **Gestalt** with a selector of gestaltOSAttr. The results from this call will tell you if the Process Manager is available, whether the _Launch trap can return to the caller, and whether the new parameters and flags for the **LaunchApplication** call are available.

▶ The Notification Manager

The Notification Manager provides a set of services that enable processes running in the background to notify the user without disrupting the process running in the foreground. Potential clients of these services include applications, desk accessories, INITs, device drivers, Time Manager tasks (which you'll look into in the next section), VBL tasks, and I/O completion routines. The communications provided by the Notification Manager are in one direction only: from the background process to the user. It does not provide a direct method for the user to talk back to the code. Applications and desk accessories running in the foreground can also use the services of the Notification Manager.

The standard example of using the Notification Manager is the Print Monitor. Rather than beeping or taking over the screen, when the Print Monitor needs to tell you that the printer is out of paper, it alternates the small icon of the Print Monitor with the Apple in the menu bar. It also places a small diamond next to the application's name in the application menu.

Applications and other types of code use the Notification Manager by giving it a *notification request*, which is immediately queued by the Notification Manager. This manager handles the request as soon as it can get to it. In other words, the notification services are asynchronous.

The Notification Manager provides three different ways to alert users.

- Polite notification—By alternating a small icon with the Apple in the menu bar. The icon will continue to be displayed until the application or desk accessory removes the notification request.
- Audible notification—By playing the system beep or another sound from an 'snd ' resource. The sound will be played one time at most.

- Alert notification—By displaying an alert dialog box on the screen, which the user must acknowledge. This alert will be displayed one time at most.

You can request to use these methods in any combination. When using any of these methods, you can also optionally request that a diamond be placed next to the name of the application or desk accessory doing the notifying. The diamond, which provides a signal to the user to bring that application to the front, remains next to the name until the notification request has been removed. This option is obviously of no use to other types of code, such as device drivers.

You can also ask the Notification Manager to execute a *notification response procedure*, which will be executed after all other notifications for this request have been performed. You shouldn't use this procedure to display anything on the screen. Use it to verify that the user has responded to the notification or to perform some function if the notification was posted. If you don't need to do anything more than remove the notification request from the queue, you don't need to write a notification response procedure. When submitting your request, you can specify that the Notification Manager remove the request automatically. The request will be removed from the queue, but it will not be deallocated.

Apple recommends that polite notification be the default. If you use audible notification, allow the user to turn it off. Apple also recommends that users be able to turn off all background notification if this is safe (if no data will be lost).

▶ Using the Notification Manager

To install a notification request, fill out a notification request and specify what method(s) of notification you'd like. You can optionally specify the address of a notification response procedure to be executed as the final stage of a notification. Then call **NMInstall** to install this record in the Notification Manager's queue. If a request is installed by an INIT, then the user will not be notified until the operating system is completely up and running.

Call **NMRemove** to remove a notification request from the queue, regardless of whether it has been processed.

▶ The Time Manager

System 7 brings the third major revision of the Time Manager. The second version of the Time Manager was introduced with System 6.0.3, but was never documented outside Apple Computer. The first version was

introduced with the Macintosh Plus and was documented in *Inside Macintosh*, Volume IV. The operating system was then the primary client of the Time Manager. Since then, some applications have used it to schedule tasks, but the third version makes it much more accurate than the previous versions.

The Time Manager provides some time-related services that enable you to accurately schedule routines to execute at a particular time, or to execute at some periodic interval. You can also use the Time Manager to accurately measure time intervals.

Important ▶ Multimedia and animation applications should use the facilities of the Vertical Retrace Manager for scheduling display activities. This can reduce or eliminate flicker, something you can't do when using the Time Manager.

The time-related services are independent of the CPU chip on the current machine. Measuring time by knowing how many clock cycles a loop of code takes to execute is no longer accurate, because the 68020 and later CPUs have an instruction cache. These CPUs can process parts of instructions before they are executed, making it difficult or impossible to know how long a set of instructions will take to execute (in terms of CPU cycles).

All of the Time Manager calls use Time Manager task records, which the Time Manager maintains in a queue. There are two versions of these records. You must use the new version of these records to get the highest accuracy from the Time Manager, but you can use either version of these records with the latest version of the Time Manager.

▶ Accuracy of the Time Manager

In the first version of the Time Manager, the time intervals could be specified in milliseconds and the maximum range was almost 24 hours. The latest version of the Time Manager allows delays to be specified to a resolution of 20 microseconds. The maximum delay specifiable in milliseconds is about a half an hour.

Use the higher accuracy available in the new version of the Time Manager to measure time intervals. Note, however, that if you try to schedule a task to run frequently, you may not have any CPU time left for anything else.

▶ Fixed-Frequency Scheduling

The previous versions of the Time Manager made it impossible to execute a task at a drift-free, fixed periodic interval, because the time count specified when calling **PrimeTime** (see below) was relative to the current time. This count couldn't take into account the execution time needed by the Time Manager and other low-level code. With the latest version of the Time Manager, when a task is rescheduled with the **PrimeTime** call, the Time Manager modifies the time count you pass to it so as to take into account the amount of time since the task last expired. This means that a task can be scheduled to execute at a fixed frequency, and the execution time won't drift.

▶ Time Manager Tasks

Time Manager tasks execute at interrupt time. This means that they cannot use the Memory Manager in any way, either directly or indirectly. Tasks also should not access handles to unlocked blocks—these may not be valid at interrupt time. The A5 world may not be valid either, so you need to set up the A5 world if you need access to application globals. Tasks should preserve the state of all registers besides A0 through A3 and D0 through D3.

When quitting, remove all tasks your application has installed. Otherwise, the Time Manager will try to execute the task. Since the pointer to the task will no longer be valid, you might cause the operating system to crash. If you want to install a task that will continue to execute after your application has quit, you could install the task in the system heap.

Use the Notification Manager if you need to communicate directly to the user from a Time Manager task. This is because the task is executing in the background (in the sense that it is "behind" the application).

▶ Compatibility and the Time Manager

If you want to use the newest features of the Time Manager, you'll need to check which version of the Time Manager is available when your application is running. Do this by calling **Gestalt** with a selector of gestaltTimeMgrVersion. If the version number is equal to (or greater than) 3, then you can use the features introduced with System 7.

▶ Creating and Using Time Manager Tasks

You can use Time Manager tasks for doing any task that must be executed periodically according to an absolute clock. This is different from tasks that must be executed periodically relative to a display's vertical retrace interval. You might use a Time Manager task to generate periodic events that force an application to update a window with the latest values from some background activity. For example, you could display accurate statistics about a communications channel or a real-time process, such as errors per second.

You must first insert a Time Manager task into the Time Manager's queue using either the **InsTime** or **InsXTime** system call. With either call, you must pass a pointer to a Time Manager task record. This record contains a pointer to the task that will be executed by the Time Manager at the specified time. Use the **InsTime** call if you only need the features of the initial version of the Time Manager. In this case, you would pass a pointer to a task record of the initial structure. If you need the new features, use **InsXTime** and pass a pointer to a new task record structure. These calls only add the task record to the queue; they do not schedule the task.

Call **PrimeTime** to schedule or reschedule a Time Manager task that has been previously queued. You need to pass a pointer to a task record (which can be either the initial or new structure) and a time count. This time count tells the Time Manager when to first execute the task. If this count is positive, the count is in milliseconds. If it is negative, it is in microseconds. If it is 0, the routine is executed as soon as interrupts are enabled.

You should remove tasks from the Time Manager's queue with the **RmvTime** call. You need to pass a pointer to the task record to be removed. Time Manager tasks are removed whether or not they've been scheduled and whether or not they have executed. This call returns the amount of time remaining before the task would have been executed in the task record. If this count is 0, the task has been executed. If the count is negative, then the number is in microseconds, and if it positive, it is in milliseconds.

▶ Measuring Time Intervals

At various times, you need to accurately measure a time interval. You need to know how long a user takes to perform a task, or how long it takes to send a message to another machine over a complex network.

You can accurately measure time intervals with the Time Manager. Do

this by installing a task with the **InsTime** call, or use **InsXTime** for the most accuracy. Call **PrimeTime** at the start of the interval to be measured, and specify a time count longer than the interval you are measuring. When the interval is over, call **RmvTime**. The task record will contain the remainder of the time before the task would have been executed. The difference between this value and the starting value tells you the elapsed time.

▶ Conclusion

In this chapter, you've looked at three managers associated with processes and multitasking on the Macintosh. Although the operating system does not yet provide preemptive multitasking, simpler forms are available today. You can now create multiple applications that can run at the same time and that can exchange messages using AppleEvents, interapplication communication, and low-level program-to-program communications. This will make it easier to develop complex applications as a set of simpler, cooperating applications.

Get Info ▶

For more information on the Process Manager, read the Process Manager chapter in *Inside Macintosh*, Volume VI. This chapter contains a more detailed explanation of how processes are scheduled and some options for controlling them. Also, refer to the Event Manager chapter in *Inside Macintosh*, Volume VI for more information about the **WaitNextEvent** call.

For more information on the Notification Manager, read the Notification Manager chapter in *Inside Macintosh*, Volume VI. Note that the APDA publication *The Programmer's Guide to MultiFinder* is obsolete with respect to System 7. You'll still need it if you're developing for System 6, however.

For more information on the Time Manager, read the Time Manager chapter in *Inside Macintosh*, Volume VI. This chapter explains in detail how you can pass a reference to your application's A5 world to a Time Manager task. For more information about the other time-related services available on the Macintosh, read the Vertical Retrace Manager chapters in *Inside Macintosh*, Volumes II and V.

18 ▶ The File System

▶ Introduction

Over the years, the file system of the Macintosh operating system has grown from the Macintosh File System (MFS) on the first version of the Macintosh operating system to the Hierarchical File System (HFS), which was introduced simultaneously with the Macintosh Plus.

Note ▶

> MFS was actually a flat-file system, although it simulated a directory system. As larger disks became more common, the performance of the MFS got worse and worse. This led to the development of the HFS. MFS is used only occasionally today, but is often seen only on 400K (single-sided) floppies.
>
> HFS brought significant improvements in performance over MFS. HFS, the predominant file system used on Macintosh computers today, is the default file system for 800K (double-sided) and 1.4 Mb (high-density, double-sided) floppies as well as for hard disks.

System 7 brings several improvements in the file system, which you'll look at in turn: file IDs, which allow applications to track files (on a single volume) over changes in name and location; FSSpec records, which simplify the task of specifying files; a single system call that searches the directory of a volume for files meeting various criteria; and the numerous new routines added to the File Manager to maintain consistency with older sets of calls.

You'll then look at some changes to the Resource Manager that affect resource files.

Then you'll look at another user-visible change in the file system: aliases. An *alias* is a file that provides a reference to another file. The alias behaves much like the file it references, but it allows the file to appear in multiple places.

Next, you'll look at the improvements to the Standard File Package. This package is used by almost all applications when a user opens an existing file or creates a new file.

Another significant change in the file system is that all the Finder-related information is now kept in a single database. Apple provides a set of calls to read from and write to this database, although applications are discouraged from writing to it. You'll look at this database and its associated system calls.

Yet another change in the file system—one that is quite visible to the user—is in the architecture of the System Folder. The System Folder is not only the place where the installed System and Finder live, but also where almost all operating-system related resources live—printer drivers, network and communications drivers, and control panel devices, for example. Other files that end up in the System Folder include INITs, preferences files, temporary files, and the downloadable PostScript font files. The architecture introduced with System 7 cleans up the clutter by providing a standard set of folders inside the System Folder for various categories of these files.

▶ Compatibility and the File and Alias Managers

The File Manager has many new calls, as described in this chapter. Verify that you can use these calls on the current volume by calling **PBHGetVolParms**. This new File Manager routine fills out an HParamBlock record, which describes the volume in detail.

The Alias Manager is available under System 7 and later versions of the operating system. To ensure that the Alias Manager is available, you should call **Gestalt** with a selector code of 'alis'.

▶ File IDs

Under HFS, applications had a difficult time keeping track of files. You could keep track of where you last saw a file because you could know what directory it lived in and its name. If the user either renamed the file, moved it to another folder, or moved the folder in which the file lived, then there was no way to relocate that file.

Important ▶ Apple recommends that you use aliases in preference to file IDs for tracking files on a volume.

File IDs, introduced as part of System 7, solve most of that problem. An application can create a unique file ID for a file, and this number will remain associated with that file until the file is deleted, the file ID is deleted, or the file ID is exchanged with one from another file. The last case sounds a little strange, but you might want to do this when the user "saves" a file so that the latest version of the file maintains the same file ID. File IDs, which are unique only within a volume, are available only on HFS volumes.

Folders are not affected by this change because folders have unique IDs, called *directory IDs*. (The terms *folder* and *directory* are synonymous.) You cannot create a file ID for a folder.

▶ Working with File IDs

File IDs are not created automatically. Your application can check whether a file ID already exists for a file by calling **PBGetCatInfo**. This call returns an HParamBlockRec record, which contains most everything there is to know about a file. The HParamBlockRec record has been enlarged with the addition of a new param block type FIDParam, which contains a pointer to the file's name, the directory ID for the directory in which the file lives, and the file ID for the file.

Call **PBCreateFileIDRef** to create a file ID. Call **PBDeleteFileIDRef** to delete a file ID. Given a file ID, the call **PBResolveFileIDRef** returns the parent directory ID and the file name for the file with that ID. Since these three routines are low-level calls, you'll have to fill out an HParmBlk, the standard HFS parameter block, when calling any of these routines.

Another new system call, **PBExchangeFiles**, lets you swap both the data and resource fork of two specified files, as well as their modification dates (in the volume catalog). Restrictions on this call are that both files must exist and be on the same volume. File IDs do not have to exist for either file, and the call works for open or closed files. This call is of special use when handling the "Save" and "Save as..." commands from the user. Backup programs should now be using file IDs in preference to file names when restoring files from the backup set. **FSpExchangeFiles** is the corresponding high-level call.

File IDs shouldn't be used as a way of specifying a file—that's what aliases are for (see below). The Alias Manager uses file IDs internally, but applications are not aware of file IDs from it.

Four existing HFS routines have been enhanced in System 7 to work with file IDs: **PBHDelete**, **PBHRename**, **PBCatMove**, and **PBGetCatInfo**.

▶ FSSpec Records

FSSpec records are the new and preferred way to specify files. Instead of using various and sundry combinations of volume reference number, working directory ID, drive number, directory ID, partial pathname, and/or full pathname, most routines that deal with files now take FSSpec records.

An *FSSpec record* consists of a volume reference number, a directory ID, and a file name. This means that you cannot use working directory reference numbers and the like to identify an object. Fortunately, the **FSMakeFSSpec** system call can take as input any reasonable combination of volume reference number, working directory ID, drive number, directory ID, full pathname, partial pathname, and file name. It returns the canonical file reference for that file, directory, or volume.

PBMakeFSSpec is the corresponding low-level File Manager call. As is usual with these low-level calls, you will have to fill out an HParmBlk record to use this call.

▶ Searching for Files

Searching for files has been simplified with the **PBCatSearch** system call. Previously, searching a volume for a set of files that met a set of criteria required an exhaustive search through the volume directory. Most searches can now be completed with less work (number of lines of code) in less time with this new system call.

The search criteria are specified in a pair of CInfoPBRec records and an ioSpecBits record that specifies which fields to use as search criteria. CInfoPBRec records are the same structures used with the **PBGetCatInfo** call. Two copies of this data structure are used by the **PBCatSearch** call: one record to specify the lower bounds for searching, the second for the upper bounds. Actually, it's a little more complicated than that, but the general idea is that you can search on almost any data stored in the file directory. This includes file type and creator, data fork length, resource fork length, and so on.

Options for the **PBCatSearch** call include the following:

- The use of a read buffer to increase performance. A buffer as small as 1K helps, although 32K is recommended for optimal performance.

- A count for the maximum number of matches to return.
- A maximum count of catalog records to search and a starting position in the catalog. This feature allows your application to check with the user every so often.

Searching through volumes is a lot easier and faster using **PBCatSearch** than with the old methods. Note the word *volume*: at this time, you cannot use **PBCatSearch** to search a folder on a volume.

▶ Other Changes to the File Manager

Call **PBGetForeignPrivs** and **PBSetForeignPrivs** to manipulate file access information on foreign (non-Macintosh) file systems that use a different model of file access, such as the A/UX file system. These low-level calls make it easier for shell programs (Finder, MPW, and so on) to support the manipulation of files on these other file systems. The new routine, **PBHGetVolParms**, tells you if the current volume uses an alternate privilege model.

System 7 brings a new set of File Manager routines that support FSSpec records instead of the previous (and complex) set of file specification parameters. These routines and their older equivalents are listed in Table 18-1.

Table 18-1. New File Manager routines and their older equivalents

New File Manager Routine	*Older Equivalent*	*Purpose*
FSpOpenRF	OpenRF	Opens resource fork of a file
FSpCreate	Create	Creates a new file
FSpDirCreate	–	Creates a new directory
FSpDelete	Delete	Deletes a file
FSpGetFInfo	GetFInfo	Gets Finder information on a file
FSpSetFInfo	SetFInfo	Sets Finder information on a file
FSpSetFLock	SetFLock	Locks a file
FSpRstFLock	RstFLock	Unlocks a file
FSpRename	Rename	Renames a file
FSpCatMove	–	Moves a file to a new directory
FSpOpenDF	OpenDF	Opens the data fork of a file

The File Manager also provides a new routine, **HOpenDF**, which is the HFS equivalent of the **OpenDF** routine. Also provided is **OpenDF**, which is similar to **FSOpen**, except that whereas **FSOpen** can open either a file or a device, **OpenDF** opens only a file. For that reason, you should use **OpenDF** in preference to **FSOpen**. There are also two low-level forms of this new call: **PBOpenDF** and **PBHOpenDF**.

Also provided by System 7 are a series of new high-level calls that match the low-level HFS calls introduced in *Inside Macintosh*, Volume IV. These are listed in Table 18-2. The new high-level calls are easier to use, since they don't require filling out a complex parameter packet, as the lower-level calls require.

Last, the version of the File Manager in System 7 includes a new set of routines, listed in Table 18-3, that use directory IDs in place of the older MFS function calls. MFS (Macintosh File System) is the original file system used on Macintosh computers. It is generally encountered only on

Table 18-2. New high-level HFS calls

New HFS Call	*Low-Level HFS Call*	*Purpose*
AllocContig	PBAllocContig	Allocates a contiguous amount of disk space
DirCreate	PBDirCreate	Creates a new directory
CatMove	PBCatMove	Moves a file to a new directory
OpenWD	PBOpenWD	Opens a working directory
CloseWD	PBCloseWD	Closes a working directory
GetWDInfo	PBGetWDInfo	Gets information on a working directory

Table 18-3. New HFS equivalent routines for the MFS routines

New HFS Call	*MFS Call*	*Purpose*
HGetVol	GetVol	Gets a volume ID
HSetVol	SetVol	Sets the current volume ID
HCreate	Create	Creates a new file
HOpen	Open	Opens a file
HOpenRF	OpenRF	Opens the resource fork of a file
HDelete	Delete	Deletes a file
HSetFLock	SetFLock	Locks a file
HRstFLock	RstFLock	Unlocks a file
HRename	Rename	Renames a file
HGetFInfo	GetFInfo	Gets information on a file
HSetFInfo	SetFInfo	Sets information on a file

400K floppy disks these days. The HFS (Hierarchical File System) is the current file system used on the Macintosh. These new routines provide an HFS equivalent for these older calls.

Three calls have been added to the File Manager that allow you to mount remote volumes without going through the Chooser. Call **PBGetVolMountInfoSize** to find out how large a record will be required to store its mounting information. Then allocate a handle of that size and call **PBGetVolMountInfo** to get the volume's mounting information. Last, call **PBVolumeMount** to mount the volume.

▶ Compatibility and the File Manager

Before using the new FSSpec routines, verify that they are available by calling Gestalt with a selector of gestaltFSAttr. This will let you know if it is safe to use the new File Manager routines, including those which use FSSpec records.

▶ Changes to the Resource Manager

The Resource Manager has been enhanced in System 7 in three ways. Numerous system icons and default icons for applications are documented in the Resource Manager chapter of *Inside Macintosh*, Volume 6. The standard resource types are also defined there.

Second, there are several new system calls that take FSSpec records or HFS data. These routines are parallel to existing Resource Manager routines. The new routines are named **FSpCreateResFile** and **HCreateResFile** (the orginal routine is named **CreateResFile**), and the other set are named **FSpOpenResFile** and **HOpenResFile** (the orginal routine is named **OpenResFile**).

Third, there are some new routines which make it easy to read and write large resources. Previously you would have to work with the entire resource in memory or write some tricky code. These new routines are named **ReadPartialResource** and **WritePartialResource**. Both routines work with a buffer that you supply. Before using either routine, first call **SetResLoad** and specify FALSE so that the Resource Manager does not try to load the entire rosource into memory. Call **SetResourceSize** to change the size of a resource on disk. This routine does not write any data to disk.

▶ Aliases

Aliases are alternate names for files, directories, and volumes that provide more flexibility for organizing the desktop. For example, some applications require several data files (and perhaps some external code in

separate files) to run. Under previous versions of the operating system, the user had to remember where all these files were stored. Under System 7, the user can keep all the files together in one folder and put an alias to the application on the desktop (or wherever else the user might keep his or her applications). This isn't an earth-shaking feature, but it's a nice feature that some people will find useful in customizing their desktop.

Aliases are also a useful way of keeping references to dynamic files that might live on an AppleShare volume. For example, a list of project codes for timesheets could be kept on an AppleShare volume. This file could be easily updated on the server, and users would not have to remember to download the file each time it was changed.

The Finder (and other applications) let a user create aliases for desktop objects (file, directory, or volume). The icon for the alias is the same icon used for the object, but the name of the alias is in italics and consists of the name of the original object with a suffix of *Alias*. The alias behaves in every other way like the original object.

The Alias Manager provides routines that allow applications, such as the Finder, to work with aliases. These routines allow your applications to track files that have been moved, copied, renamed, or restored from a backup.

Aliases are used by several components of System 7. The Finder, the most visible user of aliases, allows users to create and manipulate aliases to files. Names of files that are aliases show up in italics in the Finder. The Edition Manager uses aliases to track edition files for both publishers and subscribers. Aliases are used in AppleEvents to name files. The Standard File Package and the Finder automatically resolve aliases for applications.

You should use aliases whenever you save the location of a file or directory. Aliases should replace the use of pathnames, file names, directory IDs, and volume names when you're saving the location of a file.

▶ Alias Records, Canonical File Specifications

An alias record identifies the desktop object (file, directory, or volume) to which the alias refers. Alias records are used as parameters to the Alias Manager calls and should not be examined by applications. The only public data in alias records are the file type of the target and the length of the alias record; all other information is private and is not documented. The private information contains volume names, directory names, file names, directory IDs, file IDs, creation dates, and AppleShare information.

You can store your own application-specific data in an alias record. After bringing an alias record into memory, use the Memory Manager to increase the size of the record. Then add your data to the end of the

alias record. The Alias Manager will maintain your data at the end of the alias record even if the private contents of the record are changed by the Alias Manager. Good manners imply that you should add application-specific data only to alias records that your application has created. If more than one application tries to add data to the same record, there is no way to guarantee the integrity of the private data in the record.

Alias records exist only in memory. You must write them to disk (and read them from disk) yourself. These can be kept as resources of type 'alis'. You can use the Alias Manager routines to create alias records to existing files and to resolve alias records to the file to which the alias points. The routines of the Alias Manager require the use of FSSpec records, which were described earlier in this chapter.

▶ Calling the Alias Manager

Call **NewAlias** to create an alias record to a target file. You can optionally specify a source file if you want the alias to record a relative path. The default is to record absolute path information. Call **NewAliasMinimal** to create an alias record that records only the minimal amount of information about the target. This means that the alias record includes the volume name, volume creation date, parent directory ID, and name of the target. You'd use this call when the speed (of resolution) is more important than the robustness of the alias resolution, such as when you will be using an alias record for a brief period of time. Call **NewAliasMinimalFromFullpath** to create an alias record that records the full pathname of the target (including the volume name). You can also use this call when speed is more important than the robustness of the alias resolution—for example, when the alias record will be used for only a short time and when the target of the alias lives on a removable volume.

Two calls are used to resolve alias records: **ResolveAlias** and **MatchAlias**. Call **ResolveAlias** to get the FSSpec record for a single target or an error if the alias cannot be resolved. If the target lives on a mountable volume, then **ResolveAlias** will attempt to mount the volume. You are informed if the alias record was updated as a result of this call, but it will not be saved automatically. It's up to you to save the updated alias record. You can ask the alias to be resolved either absolutely or relatively.

MatchAlias is a low-level call that takes as input an alias record and a set of rules to control the search process. It can resolve an alias record to one or more target files in the form of canonical file specifications. You can also specify various rules and options that should be used when resolving an alias. **MatchAlias** can also be passed an optional alias-filtering func-

tion that will be called for each possible match or after some time has passed without a match.

Call **UpdateAlias** to update the specified alias record to point to the object described by an FSSpec record. You'd do this following a call to **MatchAlias** when you've been told that the alias record was changed. If you call **UpdateAlias** and specify a minimal alias record (such as a record created by either **NewAliasMinimal** or **NewAliasMinimalFromFullpath**), the alias record will be converted to a full alias record. **UpdateAlias** returns a Boolean indicating whether the alias record has been changed as a result of this call.

Some of the information in an alias record can be extracted with the **GetAliasInfo** call. You can request an object's name, volume name, parent folder name, server name, and zone name. The latter two categories are returned only if the object lives on an AppleShare volume.

The File Manager also provides a routine that is helpful when dealing with aliases. If you open files without using the Standard File Package, then it's up to your code to resolve the alias. If you don't resolve it, you'll be opening a file with no contents! Call **ResolveAliasFile**, part of the File Manager, in cases when your application receives an FSSpec record from a source other than the Finder or the Standard File Package. If the FSSpec record points to an alias, the alias will automatically be resolved for you.

▶ The Standard File Package

The Standard File Package is used by most applications to provide an interface that allows users to do the following:

- Navigate to a particular directory and select a file to open
- Navigate to a particular directory and provide the name of a file when saving a document

The Standard File Package has been improved in System 7 to take advantage of various improvements and changes in System 7. For programmers, the latest version of this package provides the following:

- A new pair of routines, **StandardPutFile** and **StandardGetFile**, to present the two standard dialogs
- Another new pair of routines, **CustomPutFile** and **CustomGetFile**, to present customized versions of the two dialogs
- A new data structure, StandardFileReply, used by all four of the new routines.

All of these routines use the FSSpec data structure, described earlier in this chapter.

The user will notice some changes in the Standard File Package. The design of the dialogs has been improved. For example, the previous versions of this package showed only one volume at a time. The System 7 version shows, at the topmost level, all objects on the desktop, including volumes, files moved to the desktop, and the Trash can. Another example: when saving a file, the user can now tab into the scrolling list of files. Any keystrokes entered while this list is the active item will scroll the list to show the first file that matches those characters.

Call **StandardGetFile** to present the user with the standard dialog box, allowing him or her to select the volume, folder, and file to be opened, as illustrated in Figure 18-1. You specify a list of file types for the files that appear in the list and an optional file filter procedure. This procedure is called for each file that matches the specified file types, and returns TRUE if the file should be displayed or FALSE if not. Rather than specify a list of file types to be displayed, you can also ask that all files be displayed; this is useful in conjunction with a file filter procedure to select files on some

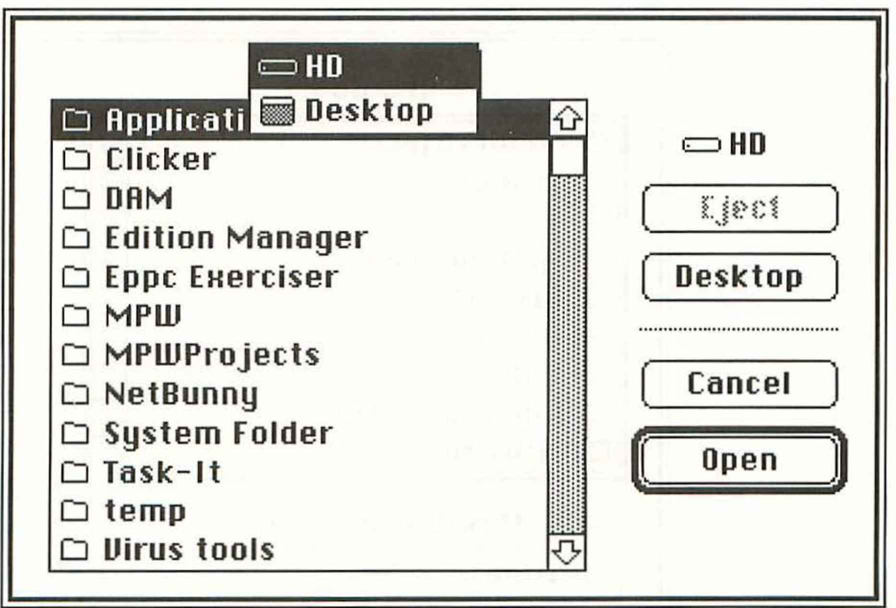


Figure 18-1. The dialog box presented by the StandardGetFile routine

basis other than simply file type. When the user presses the Open button, the **StandardGetFile** routine returns an FSSpec record, which contains the details of the chosen file. This data structure was described previously in this chapter.

Call **CustomGetFile** to present the user with a customized dialog box to select a file to open. In addition to specifying all the parameters needed for the **StandardGetFile** routine, you must also specify an optional 'DLOG' resource ID if you need a nonstandard dialog. Use NIL if you are customizing the standard dialog, or another ID if you are creating your own dialog. Among the other parameters are pointers to callback functions, including a dialog hook procedure (which handles hit items returned from the Dialog Manager), a pointer to a modal filter procedure (which filters and optionally processes events received from the Event Manager), and a pointer to an activation procedure (which controls the highlighting of any text fields defined by your code).

Call **StandardPutFile** to present the standard dialog box for requesting a name and directory when saving a file as illustrated in Figure 18-2. Specify a prompt (which is displayed at the top of the dialog box) and an

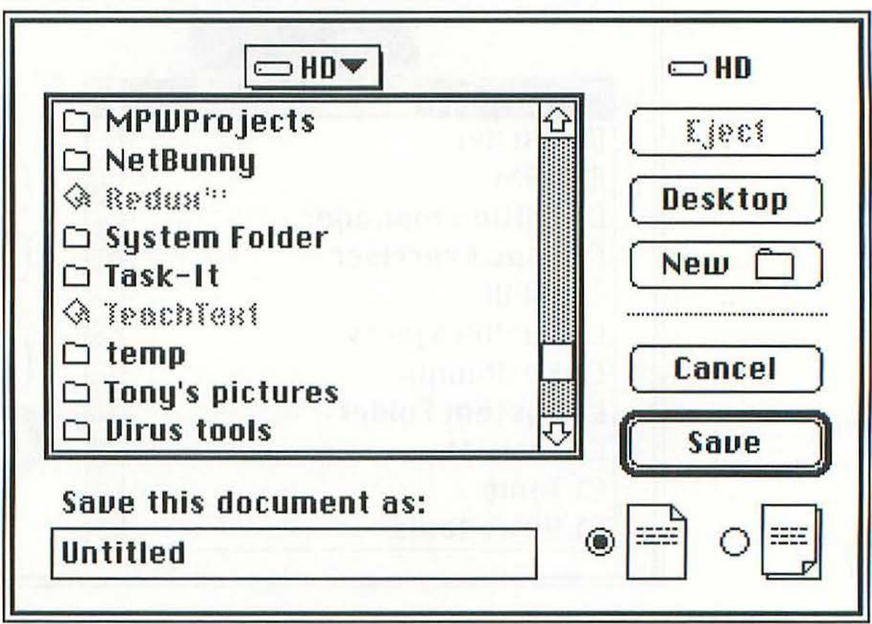


Figure 18-2. The dialog box presented by the StandardPutFile routine

optional default name. This routine will return a **StandardFileReply** record when it completes.

Use the **CustomPutFile** call to present a customized dialog box for requesting a name and directory when saving a file. In addition to the parameters mentioned above, you'll also specify a dialog ID. Use NIL if you want to customize the standard dialog, or use another dialog ID if you are creating your own dialog. Among other parameters, you'll also specify pointers to the callback functions.

▶ Compatibility and the Standard File Package

When writing code, use new routines **StandardGetFile** and **CustomGetFile** in preference to the older routines **SFGetFile** and **SFPGetFile**. The same holds true for **StandardPutFile** and **CustomPutFile** with respect to **SFPutFile** and **SFPPutFile**. Before using these four new routines, verify that they are available by calling **Gestalt** with a selector of gestaltStandardFileAttr. At the release of System 7, the only attribute returned is that these four new routines are available or not available.

But what about applications written prior to System 7? The System 7 version of the Standard File Package will use the new versions of the dialog boxes unless the calls to the older routines use incompatible callback functions. If you called the older routines, did not use a dialog hook or modal dialog filter callback function, and did not specify an alternate dialog ID, then the user will see the new dialogs when running under System 7. If the new dialogs can be used, then they return the older data structures needed for the older routines.

▶ Finder Information and the Desktop Database

Under MFS and HFS, Finder-related information about a file was stored in an invisible file called Desktop. Finder-related information includes icons, the user's (Get Info) comments, and so on. The only application that was supposed to access this file was the Finder. Unfortunately, sometimes other applications also need access to this information. The Desktop file was never documented, and it changed now and then. Application developers were forced to disassemble the file to determine its secrets. This made life difficult for developers and users, especially when a new release of the system changed some of the data structures.

Under System 7, the file system now maintains Finder-related information in a database. This database can be accessed by a new series of system calls. Now Apple provides an official way to find this information, and it won't change with minor changes in the structure of the database.

By the Way ▶ The database containing Finder-related information is not new—AppleShare used an early version of it. Since the old Desktop file kept most information in resources, and resources cannot be shared by multiple applications, the developers of AppleShare were forced to devise a new way of storing this information.

The new desktop database file is created for volumes larger than 2 Mb—basically, any volume larger than a floppy. The database file is kept on the volume itself if the disk is writable, and on the boot volume if the volume is not writable—for a CD, for example.

▶ System Calls to Access the Desktop Database

To access information in the desktop database, you need to open the database file. Do this with the **PBDTGetPath** routine. Another call, **PBDT-OpenInform**, performs the same function as the **PBDTGetPath** call, but it also tells you whether the database was already open. There is a **PBDT-CloseDown** call, but applications do not use it!

Table 18-4 lists the new system calls that enable you to read, write, or delete information from the desktop database.

Table 18-4. System calls to access the desktop database

Data	*Read Data*	*Add Data*	*Remove Data*
Icon	PBDTGetIcon	PBDTAddIcon	–
Icon description	PBDTGetIconInfo	–	–
File types supported by an application	PBDTGetAPPL	PBDTAddAPPL	PBDTRmvAPPL
User comment	PBDTGetComment	PBDTSetComment	PBDTResetComment

The icon bitmap calls—**PBDTGetIcon** and **PBDTAddIcon**—let you access the database for various types of icons, given their signature (creator and type). Remember that the Finder uses file creator and type to locate an application that can open a file with those types. Call **PBDTGetIconInfo** to retrieve icons associated with a particular application signature.

The file type calls—**PBDTGetAPPL**, **PBDTAddAPPL**, and **PBDT-RmvAPPL**—give you access to the information connecting applications and their signatures. **PBDTGetAPPL** returns the details of the application(s) associated with a particular signature.

The file comment calls—**PBDTGetComment**, **PBDTSetComment**,

and **PBDTResetComment**—give you access to the Get Info comments of files and directories. The maximum length of a comment is 199 characters. If you provide a longer comment, it will be truncated.

One other call is of importance to application writers: **PBDTFlush**, which ensures that any changes made to the desktop database are flushed from memory to disk.

There are three versions of the routines listed in Table 18-4 and of **PBDTFlush**. These routines all take a parameter that controls whether the routine will run synchronously or not. There are two other versions of each routine (such as **PBDTGetIconInfo**) in that table that run synchronously (**PBDTGetIconInfoSync**) and asynchronously (**PBDTGetIconInfoAsync**). The latter two versions of the routine run slightly faster since they do not use any glue code.

▶ New Icon Types Supported by the Finder

The Finder now supports many new types of icons for display under different circumstances. These new types are listed in Table 18-5. The four new color icons do not include a mask—they use the monochrome mask. The resource ID of the 'ICN#' governs the resource ID of all other icon types.

Table 18-5. Icon types supported by the Finder

Resource Type	*Size (pixels)*	*Contents*
'ICN#'	32 by 32	Large monochrome icon and mask
'ics#'	16 by 16	Monochrome icon and mask
'icl4'	32 by 32	Large 4-bit color icon
'ics4'	16 by 16	4-bit color icon
'icl8'	32 by 32	Large 8-bit color icon
'ics8'	16 by 16	Small 8-bit color icon

Speaking of icons, you may also want to create icons for edition files created by your application and, if your application supports stationery, for stationery documents.

▶ Document String Resources Supported by the Finder

The Finder, if it cannot find the creator application for a document when the user tries to open or print a document, now looks for string resources in the document before displaying the default alerts. Your program name

should be stored in documents as 'STR ' resource ID −16396. The Finder will use this string when displaying the appropriate alert (such as *This document could not be opened...*).

A message explaining why the user cannot print or open a file should be stored in 'STR ' resource ID −16397. You should include this resource in files, such as preferences files, that the user shouldn't open or print. If the Finder finds a string resource of this type, it will display this message rather than its default message.

▶ Architecture of the System Folder

The System Folder, which was originally intended to hold little more than the System and Finder, has been used to store increasingly more files. It is not uncommon to find over a hundred files in this folder. System 7 introduces an architecture to create some order out of this chaos.

System-related folders were introduced with System 7. They are all located inside the System Folder unless noted otherwise in the following list:

- *Preferences*—Contains application preference files. Information that will be shared over a network should not be stored in this folder.
- *Temporary Items*—Contains temporary files created by applications. This folder is invisible to users and is located at the root level of the volume.
- *Extensions*—Contains INITs, printer drivers, AppleShare, and other code that provides system services.
- *Apple Menu Items*—Contains desk accessories, applications, files, and folders that the user wants to see in the Apple menu. When the user selects an item from the Apple menu, the Finder opens that item.
- *Control Panels*—Contains cdevs (control panel devices).
- *Startup Items*—Contains applications and desk accessories that the user wants launched at system startup time.
- *PrintMonitor Documents*—Contains spooled documents awaiting printing.
- *Desktop Folder*—Contains the contents of the desktop. This allows the desktop to behave like just another folder. This folder is invisible to users on the desktop, but appears at the root in the standard file dialogs.

- *Trash*—Holds files and directories that the user has placed in the Trash can. The trash is emptied only on the user's request. This change makes the Trash can behave like any other folder. If the Macintosh is running in a shared environment, then there is a shared Trash folder, which contains a separate Trash folder for each authorized user. This folder appears to users at the root of the file system in the standard file dialogs.
- *System*—This file also contains the system software. It behaves as a folder with respect to fonts, sounds, and other movable resources.
- *Communications Folder*—Contains connection, file transfer and terminal tools for use by the Communications Toolbox.
- *Rescued Items from volume name*—Contains items which were in the Temporary Items folder prior to a system crash.

A volume can have at most one of each of these system-related folders. The names of these folders are internationalized, so they are different in different countries. However, each of these folders has a different file type, so they are uniquely identified no matter which language is used to name them. Now let's look at how this architecture affects application software.

Of these system-related folders, only the Preferences and Temporary folders are normally of interest to applications. For the most part, the user and the Installer only modify the contents of the other folders.

▶ Locating the Preferences and Temporary Folders

You can locate the Preferences or the Temporary folder (or any of the other special folders) by using a new system call, **FindFolder**. You pass the folder type of the folder you're trying to find to the call, and it returns the volume reference number and directory ID for the folder. You can also specify whether the folder should be created if it does not exist.

The Preferences and Temporary folders are the obvious place to create and store a preferences file and any temporary files that your application needs. If you create a temporary file, be careful to delete it when it is no longer needed. If the application does not delete it, the space will not be reclaimed because the user won't be able to see the temporary file, let alone delete it.

Before using the new **FindFolder** routine, verify that it is available by calling **Gestalt** with a selector of gestaltFindFolderAttr. At the release of System 7, the only attribute returned is that this new routine is available or not available.

▶ Conclusion

In this chapter, you've looked at the changes System 7 has brought to the File Manager. The most important changes for application writers are the following.

- File IDs—Provide a "sticky" way to uniquely identify files on a volume
- FSSpec records—Simplify the specification of files
- The **PBSearchCat** system call—Provides a fast, controllable method of searching for files with a variety of criteria on a volume
- Many new calls to the File Manager—Maintain consistency between the various sets of File Manager calls

You've also looked at aliases, a new operating-system feature added with System 7. Aliases provide a way to give additional names to existing file system objects.

The Standard File Package has been improved, with several new routines that provide access to the new dialogs. Customizing these dialogs is easier using these new calls.

The Finder maintains file-related information in a database. Applications now have access to this information through a new set of routines.

Get Info ▶

For more information on the File Manager, refer to the File Manager chapter of *Inside Macintosh*, Volume VI. You may also want to refer to the File Manager chapters of *Inside Macintosh*, Volumes I, IV, and V.

For more information on the changes to the Resource Manager, refer to the Resource Manager chapter of *Inside Macintosh*, Volume VI.

For more information on the FinderInterface, refer to the Finder Interface chapter of *Inside Macintosh*, Volume VI.

For more information on the Alias Manager, refer to the Alias Manager chapter of *Inside Macintosh*, Volume VI.

For more information on the Standard File Package, refer to the Standard File Package chapter of *Inside Macintosh*, Volume VI. You may also want to refer to the Standard File Package chapters of *Inside Macintosh*, Volumes I and IV.

19 ▶ The Hardware Managers

▶ Introduction

In this chapter, you will look at two managers associated with managing the Macintosh hardware, both of which were revised for System 7.

You'll first look at the changes that System 7 brings to the Slot Manager. This manager provides a standard interface to talk with NuBus cards and is therefore primarily of interest to programmers writing NuBus device drivers and other low-level code. Applications should never need to talk directly with a NuBus card, and so it is covered only briefly in this book.

Then you'll move on to the Power Manager, which was introduced with System 6.0.4 and the release of the Macintosh Portable. The Power Manager provides control over the state of the electrical power to the computer for applications and drivers. It is primarily of interest to programmers writing device drivers for the Portable, so it is covered briefly.

▶ The Slot Manager

The Slot Manager provides a programming interface to NuBus cards. The routines that make up this manager are of interest to programmers writing device drivers. Occasionally, applications need to directly talk to NuBus cards, but this is unusual. The safest way for applications to talk with cards is through a device driver. To talk with a card directly, the application must change when the hardware changes. The purpose of a device driver is to insulate applications from these kinds of changes.

The changes to the Slot Manager in System 7 are primarily changes to handle cards that need to be addressed in 32-bit mode. This version of the Slot Manager was available in late versions of System 6. The previous version of the Slot Manager could talk with cards only in 24-bit mode.

▶ Slot Manager and the Initialization Process

When the Slot Manager is initialized as part of the system initialization process, it looks for an *sResource* (slot resource) on each card. This resource tells the Slot Manager where to find the driver for its card and so on. While checking for a card in each NuBus slot, the Slot Manager assembles a *Slot Resource Table,* which contains references to all the sResources it finds.

After the Slot Resource Table has been created, the Slot Manager initializes the parameter RAM on each NuBus card and executes the initialization code for the card. This code is in a PrimaryInit record, the location of which is stored in the sResource.

The initial version of the Slot Manager in early members of the Macintosh II family could only address boards in 24-bit mode. Many new NuBus cards need to be addressed in 32-bit mode, and the new Slot Manager can do this. It does complicate the initialization process, though. Let's look at how the initialization process happens now. Two cases can happen under System 7: The new Slot Manager is in ROM or the old Slot Manager is in ROM.

The first case, where the new Slot Manager is located in ROM, has just been described. The new Slot Manager can initialize cards in 32-bit mode or 24-bit mode.

The second case, when an old version of the Slot Manager is in ROM, is more complicated. The old Slot Manager initializes all the NuBus cards that can be initialized in 24-bit mode. After the operating system is installed and patches to the system are installed, the RAM-based version of the Slot Manager is called. This manager looks at the NuBus cards a second time, looking for cards that need to be initialized in 32-bit mode. When it finds such a card, the manager adds its sResource to the Slot Resource Table, initializes the parameter RAM, and executes the PrimaryInit (initialization code).

In either case, following this phase of initialization, the Slot Manager looks through all the NuBus cards again, this time looking for SecondaryInit records, which have the same structure as PrimaryInit records. SecondaryInit records allow cards to be initialized later on during the system initialization process. The new Slot Manager allows you to avoid initializing a card until the secondary initialization. This is useful, for example,

because a 32-bit video card should be initialized after 32-bit QuickDraw is installed.

Compatibility and the Slot Manager

System 7 introduces a new call, **sVersion**, which returns the version of the Slot Manager. The new RAM-based version of the System 7 Slot Manager returns 1, and the new ROM-based version returns 2. Previous versions of the Slot Manager do not recognize this call and return a non-fatal error.

Gestalt now provides a simple way of finding what NuBus slots, if any, exist on the current machine. Call **Gestalt** with a selector of gestalt-NuBusConnectors. A bit is set for each NuBus slot that exists on the machine. This therefore tells you the slot addresses available. Note that this does not tell you if cards are installed in the slots.

New Slot Manager Routines

The new and revised routines of the Slot Manager allow you to perform four new functions.

- Disable a NuBus card during the primary initialization process and then reenable the card during the secondary initialization process
- Search for disabled sResources
- Enable and disable sResources
- Restore an sResource that was deleted from the Slot Resource Table (for example, one that was deleted during the primary initialization process)

Once again, these routines are primarily for device drivers and not for applications.

The Power Manager

The Power Manager gives applications and device drivers some control over the electrical power and related functions on the Macintosh Portable. This is needed because the Portable is always running off its internal battery or off a plug-in current. The Portable does not have an on/off switch.

The Portable has three power states: active, idle, or sleep. The latter two are low power-consumption states. In the idle state, the CPU clock

has been slowed to 1MHz. In the sleep state, the power has been shut off for the CPU, RAM, ROM, and all peripherals. When the user selects Shut Down from the Special menu, the Portable enters the sleep state. Note that the Portable is always electrically "on"; even the sleep state consumes a little power. Applications can remain open during any of these three states, although currently some applications do have problems in the idle or sleep states.

Your code can control the Power Manager by means of the sleep queue, which contains a list of procedures to be called before the Portable can be put to sleep or before it is fully awakened. These procedures perform whatever is required to prepare for the sleep state.

You can use three types of sleep requests and demands: conditional and unconditional sleep demands, and sleep requests. During a sleep request, the Power Manager checks with all other procedures in the sleep queue to see if all of these routines will accept the sleep request. If all of them do accept and the network (AppleTalk) drivers also accept, then the Power Manager walks through the queue a second time and sends each routine a sleep demand. If any of the routines denies the sleep request, then the request is canceled.

On a demand to sleep, each routine prepares for the sleep state as best it can. A conditional sleep demand can be canceled only if the network drivers don't want to sleep. In this case, the user is presented with a dialog box allowing him or her to decide whether the machine should enter the sleep state. Your drivers and applications cannot deny a sleep demand. They must prepare for the sleep state as expeditiously as possible.

When the machine wakes up, each routine in the sleep queue is called to restore state.

You can use the Power Manager to do the following:

- Request or demand that the computer be put into the sleep state
- Enable, disable, or delay the transition to the idle state
- Add an entry to or remove an entry from the sleep queue
- Get the current clock speed
- Check on the status of the battery and the battery charger
- Control power to the internal modem and serial ports
- Enable, disable, or check on the status of the wakeup timer (which controls a timer that awakens the machine at the specified time)

The Power Manager is primarily of interest to programmers writing device drivers. Some applications may be affected by the transition to the

idle or sleep state, and these applications may also need to use the Power Manager. You could also use this manager to write Portable-specific applications or utilities.

▶ Conclusion

In this chapter, you've looked at changes to two low-level managers: the Slot Manager and the Power Manager. The Slot Manager provides access to NuBus cards. Improvements to this manager provide for talking with cards in 32-bit mode, in addition to the previously supported 24-bit mode.

The Power Manager provides access to and control of electrical power on the Macintosh Portable. This manager is primarily used by device drivers, but some applications peculiar to the Portable may need to use it as well.

Get Info ▶

For more information on the Slot Manager, refer to the Slot Manager chapters of *Inside Macintosh*, Volumes V and VI. You will probably also need to read *Designing Cards and Drivers for the Macintosh II Family* (Addison-Wesley, 1990), and the Device Driver chapters of *Inside Macintosh*, Volumes II, IV, and V.

For more information on the Power Manager, refer to the Power Manager chapter of *Inside Macintosh*, Volume VI.

20 ▶ Object-Oriented Programming and System 7

▶ Introduction

Even before System 7 arrived, the Macintosh operating system was considerably more complex than, say, MS-DOS. This was true for both the total number of system calls and the level of sophistication of the services provided by the operating system. Consequently, developing Macintosh application software has always required a higher skill level from programmers than does developing software for MS-DOS or similar microcomputer operating systems.

System 7 increases the level and number of operating-system services. This won't make life any easier for programmers. In this chapter, you'll look at an important programming method that can make life easier for programmers: object-oriented programming. You'll then look briefly at MacApp, which is Apple's premier object-oriented framework for developing applications.

By the Way ▶

The one "advantage" of using *object-oriented* programming rather than *object* programming is that more bad puns and acronyms are possible with the former.

▶ Object-Oriented Programming

Programming is, for the most part, no longer about minimizing bits and microseconds. Although programmers cannot be profligate in using resources such as memory and CPU cycles, other concerns are now more

important. One of the most important problems is how to provide full-featured applications that are easy to learn and to use, but still provide access to the full power in the application. Many of these applications have twice as many, five times as many, or even ten times as many features as their equivalents from ten years ago. Spreadsheets and word processors provide some of the more dramatic examples. For example, contrast the first version of MacWrite with the latest version of Microsoft Word.

How can developers provide all these features to users and not get bogged down by all the details? The fundamental problem here is managing complexity. System 7 offers a prime example of the complexity with which programmers must deal. The original version of the Macintosh operating system had an order of magnitude more system calls than did MS-DOS. Even then, the Macintosh operating system offered roughly the same number of services as typical minicomputer operating systems such as VAX/VMS and UNIX. System 7 adds an enormous number of system calls to the Macintosh operating system. Future versions can be expected to add even more. What's a programmer to do?

Another increasingly common problem is that as applications grow larger and larger, it becomes increasingly more difficult for one programmer to understand it all, let alone do it all. The promise of object-oriented programming is that it provides some technology to help programmers manage complexity. It is most definitely not going to cure all software ills, but it is proving helpful to those who have adopted it.

Two concepts provide the foundation for object-oriented programming: objects and inheritance. An *object* provides a framework that holds a data structure and the procedures that know anything about this data structure. Object-oriented programming therefore means structuring software primarily around data and not, as is the case with procedural programming, around the processes to which you subject the data. Objects are said to contain *instance variables* (data structures) and *methods* (procedures that operate on the data structures). Well-designed objects encapsulate the data, so that if the internal structure of the data changes, ideally, only its associated methods (and no other code anywhere else) will change.

Inheritance provides a powerful mechanism for reusing code. The idea behind inheritance is that you can create new kinds of objects by describing how they differ from an existing object. The differences might include new data structures, new methods, or different behavior from a method defined in the original object. A new object is said to inherit from its ancestral object.

Object-oriented programming requires a new way of doing things.

Software design is different when you program with objects rather than with procedures. Also, object-oriented programming uses an object language, an application framework, and a development system that supports objects. This triad is illustrated in Figure 20-1. Each element in the triad is related to the other two elements. The programming language is used to write the application framework and your own code. The application framework provides a platform that implements the standard behaviors required of a Macintosh application. The development environment provides tools that understand objects.

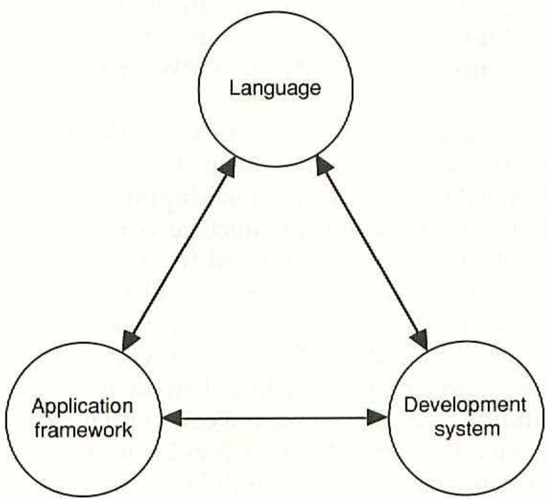


Figure 20-1. The object triad

To use object-oriented programming, you do not have to start learning everything from scratch again. You do, however, have to learn a new way of thinking about how applications are designed and implemented. The techniques of designing with objects are different from designing with procedural languages such as Pascal or C. Implementation techniques are also different. Much of what you already know is still applicable, though.

Of course, object-oriented programming isn't the only way to program. You have access to all the features of System 7 from C, Pascal, assembly language, Modula-2, Fortran, and other procedural languages. If you are working on a revision of an existing application, you'll probably continue working from the last revision. Object technology cannot easily be retrofitted into software developed using procedural languages.

Note ▶ The term *object-oriented programming* is used in contrast to *object-based programming*. Object-based programming describes programming in languages that provide objects, but not inheritance. Examples of such languages include Ada and CLU. Object-oriented languages, on the other hand, provide both objects and inheritance.

▶ What Is MacApp?

MacApp is an applications framework written in Object Pascal. It is not a programming language, although it can be used with either Object Pascal, C++, or Object Modula-2. It is not a development system; in fact, it can be used with either the Apple MPW or Symantec's THINK Pascal environment.

MacApp is a mature application framework. Apple has been working on it since 1985, and Apple has been doing work on object technology since 1982. MacApp 2.0, the version shipping at the time this book was written, offers an improved architecture over the original version. The architecture of version 2 was derived from many suggestions from programmers who used the earlier versions of the framework.

MacApp 2.0 ships with two tools that make life easier for object programmers: ViewEdit and MacBrowse. ViewEdit allows you to draw complex windows and dialogs and to get them up and running quickly. A complex window that might take a week to do in Pascal might take a day to do using ViewEdit and MacApp. MacBrowse is a source-code browser with some editing capabilities. It will help you answer questions such as "Show me all the methods and instance variables for this object" and "Show me all the objects that use this method." It provides an important navigational tool for working in the many megabytes of source code.

More and more commercial and in-house applications are being written using MacApp. Some of the commercial applications that were shipping when this book was written include the following:

- Odesta: GeoQuery—Database query program
- SoftQuad: Author/Editor—SGML structured text editor
- Analyst Workbench Products: Data Modeler—CASE tool
- Data Translation: PhotoMac—24-bit color separation software
- Icon Technology: Formulator—WYSIWYG mathematics word processor

- Apple: MacTerminal version 3—Communications program
- DigiDesign: Q-Sheet—Real-time music program
- Olduvai: Read-It!—Optical character recognition (OCR) program
- Adobe Systems: PhotoShop—Color image processing software

By the Way ▶

You can tell if an application was written using MacApp by looking at its "About..." box. One of the terms of the MacApp licensing agreement is that developers must mention MacApp in the "About..." box and manual.

Version 3.0 of MacApp provides support for many System 7 features such as AppleEvents and the Help Manager. Apple is strongly committed to updating MacApp as the operating system evolves. If you're using MacApp, this makes your job as an application programmer easier. Instead of having to make numerous nontrivial changes to take advantage of the evolving operating system, you can take advantage of all the hard work that the MacApp programmers have put into it.

▶ But What about Performance and Memory?

The traditional excuses for not using object-oriented programming are that it uses too much memory and too much disk space, and that it's too slow for anything but prototypes. These objections are based on old stereotypes of Smalltalk and LISP. This just isn't true anymore—not for Smalltalk or LISP, and especially not for Object Pascal and C++.

The amount of memory required by an application written with MacApp using Object Pascal and/or C++ to run is no more than that required by an application written using C or Pascal. This is because MacApp is exceptionally well-written code and because the compilers for the two object languages are good.

The amount of disk space required for a MacApp application is the same as for an application written in a procedural language. The only time this rule is not true is if the application is tiny. The MacApp-based application will be larger than an application written in a procedural language, unless you do some additional work to reduce the amount of code linked into the application.

Last, speed is not an issue for MacApp applications any more than it is for applications written in a procedural language. This is because compilers and linkers are good at producing high-quality code. You could write

slow code, but you can do this in a procedural language, too. Unless you are writing "hard" real-time code, there is probably no reason why you couldn't use MacApp.

The traditional excuses for avoiding object technology are no longer valid. If you want to continue avoiding it, you'll have to find a new excuse.

▶ When to Use MacApp

However, MacApp is a tool to consider if the following conditions hold true:

- You're writing an application that deals in complex data relationships—MacApp is especially suitable for "traditional" Macintosh applications such as MacWrite and MacPaint. In these applications, there is a simple, one-to-one relationship between a file, a document, and a window. Applications with more complex relationships, such as accounting packages and databases, require more work on your part than these previous types of applications.
- You're writing an application that requires real-time performance—MacApp has been successfully used for some "soft" real-time applications, but it is not suitable (with version 2.1) for use in developing "hard" real-time applications.

MacApp is the tool of first choice if you're writing an application from scratch and you don't have "hard" real-time requirements.

▶ When Not to Use MacApp

Although MacApp isn't useful for everyone, it's still easier to explain when not to use MacApp than to explain when to use it. If you're in any of the following situations, then MacApp is probably not the best choice:

- You have an existing application written with a procedural language—The architecture of applications written using MacApp is so different from that of applications written in procedural languages (C, Pascal, assembly language, and so on) that it's just not possible to retrofit MacApp into such applications. If your existing code is primarily a computational engine, however, you can put a MacApp wrapper around it to handle the user interface and input/output.
- You're writing device drivers and other low-level code—Although you can write drivers using C++, you cannot write low-level sys-

tem code using MacApp. MacApp is an *application* framework, not a generic framework for every type of code.

- You're writing an application that will be ported to other operating systems and other user interfaces—The MacApp architecture was designed solely for the Macintosh.

▶ Resources for MacApp Programmers

MacApp is available from APDA, the Apple Programmer's and Developer's Association. This is a mail-order house, run by Apple Computer, that sells all of Apple's developer tools and publications and many third-party developer tools. To purchase anything from APDA, you must join by filling out an application and paying an annual fee. Because APDA sells beta versions to get the tools out to developers as quickly as possible, this application and annual fee are required. As soon as a released version of a tool is available, APDA stops selling the beta version and sells only the final version. Usually, APDA automatically sends you the final version of a product if you've purchased a beta version.

If you distribute an application written with MacApp, you'll also have to sign a licensing agreement with the Software Licensing department at Apple. This is true whether the application is a commercial application, an internal application, or shareware. The licensing fee is small ($10/year for shareware, $100/year for everything else) and covers all of your MacApp applications.

If you are not interested in any beta versions of tools, you can alternatively purchase the final versions of tools through Developer Express. You need not sign an agreement or pay an annual fee to purchase through this program.

Another important resource is MADA, the MacApp Developer's Association. This nonprofit group was formed by developers and programmers who use MacApp. The group publishes a newsletter, Frameworks, which appears six times a year. Several products (including source code for most) are available for both members and nonmembers. Last, the group holds an annual MacApp Developer's Conference, which runs for five days and includes several days of tutorials and workshops followed by the general sessions. For more information about this group, you can write to MADA, P.O. Box 23, Everett, WA 98203, or send an AppleLink to MADA.

▶ Conclusion

Object-oriented programming is where all Macintosh developers are headed. It isn't going to solve all of our software problems, but it is a step in the right direction. Just as *structured programming* was the buzzword of the 70s, *object-oriented programming* will be the buzzword of the 90s. Even if you can't use it on your current project, learn about it now. You'll be using it in the future.

MacApp is one of the most reasonable ways to develop software on the Macintosh. Programmers maintaining existing applications might have little reason to use it now, but almost all programmers working on new applications should consider MacApp.

Get Info ▶

This chapter has provided a deliberate simplification of the issues in object-oriented programming. For an excellent introduction to object-oriented programming and MacApp, look for *Programming with MacApp* (Addison-Wesley, 1990) by David Wilson, Larry Rosenstein, and Dan Shafer. This book is available in both Object Pascal and C++ flavors, and is part of the Macintosh Inside Out series.

21 ▶ The Future

▶ Introduction

In this chapter, you'll look at three aspects of the future of the Macintosh operating system. First, you'll look at operating-system features that have been previously mentioned by Apple as candidates for inclusion in System 7. Then you'll look at features mentioned (also by Apple) as future capabilities of the operating system. Last, you'll look at where the Macintosh operating system is heading in the long term.

▶ Some Future Features

Apple introduced the concepts behind System 7 at the 1989 Worldwide Developer's Conference, held in San Jose, California, in May 1989. At this conference, Apple went through a long list of enhancements and additions to the Macintosh operating system.

The first widely distributed version of System 7 (version 7.0a9) was handed to developers at the 1990 Worldwide Developer's Conference, held in San Jose, California, in May 1990. During this one-year period, quite a few features announced in 1989 didn't make it into System 7. This was done to make it easier to ship System 7, and most or all of these features will probably be added in later versions of the Macintosh operating system. This section covers the features that missed the System 7 boat.

▶ New Print Architecture

The print architecture of the Macintosh operating system is badly in need of repair. Printer drivers were never (and still are not) fully documented, so writing and maintaining the code for a new printer (or other output device) on the Macintosh is a grueling task. A *new print architecture* is on its way. It will provide background printing, color support, and halftone support for all output devices. Documents will be independent of the output device, so a spooled file can be redirected to another type of printer without going back to the application. Custom page sizes will be much easier to create by users, applications, and drivers. A single document could contain more than one page size and more than one page orientation. The new architecture will also make it considerably easier to write printer drivers.

▶ File System Manager

A *File System Manager* will provide support for non-Macintosh file systems, such as MS-DOS, OS/2, ProDOS, and UNIX. Users will be able to mount disks formatted with non-Macintosh file systems. This manager will also allow a Macintosh to initialize a volume with one of these foreign file systems. These capabilities will clearly make it a lot easier to exchange data with other platforms.

▶ Layout Manager

The *Layout Manager* will be introduced to provide powerful text layout services. It will be much more powerful than TextEdit, which provides more rudimentary services. The Layout Manager will provide typographic-quality output by providing such features as justification, fractional positioning of characters, kerning, and ligatures. It will also directly support the contextual forms required for some non-Roman scripts. This will make it easier for programmers to develop international software.

▶ Revised SCSI Manager

A new version of the *SCSI Manager* will be introduced. It will feature support for asynchronous operations and support for direct memory access (DMA). It will also support the disconnection and reselection of SCSI devices. The new SCSI Manager will provide an implementation much closer to the ANSI standard than the previous version.

Definite Futures

Various people from Apple Computer have talked at developer's conferences about several other capabilities to be added in the Macintosh operating system.

AppleScript

AppleScript is a scripting language that will allow users to control their applications and the system. It will be based on HyperTalk, interapplication communications, and AppleEvents.

Apple Event Object Model

To provide a consistent architecture for using Apple events, both at the programmer and user (AppleScript) levels, a model for Apple events is required. The content of an Apple event is a request to perform some action on an object. The Apple event object model will describe how objects are both structured and accessed. Objects belong to a class. Objects have a set of properties, which define its nature, and a set of elements, which define its content.

Memory Protection and Multiple Address Spaces

Memory protection and multiple address spaces would keep the system and each process (application or desk accessory) in its own memory space. Code in one partition would be prohibited from reading from or writing to any other partition's memory. This would prevent one application from crashing the system or another application.

Preemptive Multitasking

Preemptive multitasking will provide more powerful multitasking capabilities than System 7. Under System 7, MultiFinder was completely absorbed into the operating system, but it still provides only cooperative multitasking capabilities. Every application that runs under System 7 depends on every other application to be well-behaved. Background tasks must be good citizens and cannot use much CPU time when the foreground application is running. These capabilities provide most of the multitasking that's needed for most of the functions that users need today, such as background printing and long calculations. A more powerful form of multitasking is needed for more powerful kinds of applications. As an example,

consider database applications. By having preemptive multitasking, you'd be able to separate the front end (the user interface), which operates at a human time scale, from the back end, which needs all the CPU time and resources it can get when processing a request.

▶ Multimedia Support

Multimedia support will be added to the Macintosh operating system. This will mean that video, audio, and animation capabilities will become much more available than they are today. Three-dimensional graphics will be part of this. Currently, QuickDraw provides drawing capabilities in two dimensions only. To present images of three-dimensional objects today, you have few alternatives except to develop the three-dimensional graphics routines on your own. This new multimedia architecture is called *QuickTime*, by analogy with QuickDraw. QuickTime will offer the following features.

- Support for data types such as video and sound, such as real-time compression and decompression
- A media-compatible document architecture that can deal with multiple forms of data in a document, including text, graphics, animation, video, and sound (on other platforms, this would be called a compound document architecture)
- Support for sound input and text-to-speech output
- Human interface guidelines for dealing with new media

▶ Network Booting

Network booting will also be supported by the operating system. This will allow a Macintosh to be booted using no local disks. This capability will be useful primarily to larger organizations. Occasional users who have a machine that boots over the network will use applications stored on a server. The process of ensuring that all users have the latest version of applications will be simplified by using this technique.

▶ Indefinite Futures

Other features are presumably going to be added someday. These features have been mentioned by Apple Computer at developer's conferences, but few details have been discussed. When these changes will occur is not clear, but their benefits are clear.

▶ Revised QuickDraw

One important change that needs to happen is improvements to QuickDraw, the drawing engine that is the foundation of the Macintosh operating system. Many applications assume that all displays use the standard screen resolution of 72 dots per inch. Every Macintosh shipped so far and all third-party video cards and monitors assume a screen resolution of 72 dpi. Higher screen resolutions are now possible in hardware, but the operating system needs to be updated to support other resolutions. QuickDraw also needs to be extended to offer functions that are not available today, such as rotated text and graduated fills. To keep the Macintosh competitive with other platforms, Apple must improve QuickDraw.

▶ Object-Oriented Programming

The other important direction for the Macintosh operating system is integration with object-oriented programming tools. Object-oriented tools such as MacApp, Object Pascal, and C++ are accelerating application development today. Apple needs to improve and develop new object-oriented development tools and environment to remain competitive with other platforms. Most of the rest of the industry is adopting object-oriented programming. Although Apple has been using object-oriented technology longer than most, Apple must take the next step and integrate the operating system with object-oriented technology. This will make it substantially easier for programmers to develop new applications by reusing existing code and by simplifying the cumbersome development process that all developers must follow today.

▶ Conclusion

In this chapter, you've looked at future goodies anticipated for the Macintosh operating system. Apple must continue to broaden and deepen the Macintosh operating system if the Macintosh is to remain a competitive platform, and the 1990s should see some spectacular innovations from Apple.

The software that we have today will change—it has to, because more and more applications are reaching the limits of complexity that our current tools can build. The future software worlds will provide us with an architecture and tools to build far more complex application systems. Instead of creating monolithic applications that try to do everything, but that do nothing well, developers will be connecting smaller applications with interprocess communications.

Index

16-bit direct color, 262
24-bit memory map, 289
32-bit cleanliness, 11, 289–292
  A5 world, 292
  compatibility, 292
  memory addresses, 291–292
32-bit memory map, 290
32-bit QuickDraw, 262–270
  See also QuickDraw
  direct color devices, 262
    PICT2 format, 265–266
    PixMaps, 263–265
  gray-scale support, 268
  improvements, 263
  indexed color devices, 263
  regions, 267
  system calls, 267–268

A

A/UX
  32-bit addresses, 290
  32-bit clean code, 59
  applications, 59–61
  compatibility, 51, 59
  managers available, 57–58
  System 7, 56–61
  Toolbox, 56–58
A/UX Toolbox, 57
A5 world, 292
AcceptHighLevelEvent system call, 74–75
Action atoms, 41
ActivatePalette system call, 272
addAssertion clause, 43–44
addAuditRec clause, 43
addPackages clause, 43
Address descriptor record (AEAddressDesc), 79
addUserDescription clause, 43
Adobe Type Manager (ATM), 106
ADSP (AppleTalk Data Stream Protocol), 248, 253–257
  Connection Control Block (CCB), 253
  Connection Listener (CL), 254, 256–256
  connection modes, 254
  data structures, 253
  system calls, 254–257
AECoerceDesc system call, 81
AECoercePtr system call, 81
AECountItems routine, 82
AECreateAppleEvent system call, 82
AECreateDesc system call, 82
AECreateList system call, 82
AEDeleteItem routine, 83
AEDeleteKeyDesc routine, 83
AEDeleteParam routine, 83
AEDisposeDesc routine, 81–82
AEDuplicateDesc system call, 82
AEGetArray routine, 82
AEGetAttributeDesc routine, 82
AEGetAttributePtr routine, 82
AEGetCoercionHandler routine, 81
AEGetEventHandler system call, 80
AEGetInteractionAllowed system call, 84
AEGetKeyDesc routine, 82
AEGetKeyPtr routine, 82
AEGetNthDesc routine, 82
AEGetNthPtr routine, 82
AEGetParamDesc routine, 82
AEGetParamPtr routine, 82
AEGetSpecialHandler system call, 81
AEGetTheCurrentEvent system call, 80
AEInstallCoercionHandler routine, 81
AEInstallEventHandler system call, 80
AEInstallSpecialHandler system call, 81
AEInteractWithUser system call, 84
AEProcessAppleEvent routine, 80
AEPutArray routine, 83
AEPutAttributeDesc routine, 83
AEPutAttributePtr routine, 83
AEPutDesc routine, 83

AEPutKeyDesc routine, 83
AEPutKeyPtr routine, 83
AEPutParamDesc routine, 83
AEPutParamPtr routine, 83
AEPutPtr routine, 83
AERemoveCoercionHandler routine, 81
AERemoveSpecialHandler system call, 81
AEResetTime system call, 84
AEResumeTheCurrentEvent system call, 81
AESEnd system call, 83
AESetInteractionAllowed system call, 83
AESizeOfAttribute routine, 82
AESizeOfKeyDesc routine, 82
AESizeOfNthItem routine, 82
AESizeOfParameter routine, 82
AESuspendTheCurrentEvent system call, 81
AFP (AppleTalk Filing Protocol), 257
Alert notification, 302
Alerts, help messages, 176–177
Alias file, 308
Alias Manager, 314–316
  alias records, 314–315
  calling, 315–316
  canonical records, 314–315
  compatibility, 308
Alias records, 96, 314–315
Aliases, 13, 23–24, 313–316
  deleting, 24
  displaying, 23
  opening files, 23
'alis' data type, 94
'alis' resource, 315
AllowPurgePixels system call, 275
AMU chip, 14
AnimateEntry system call, 273
AnimatePalette system call, 273
AppendDITL system call, 242
AppendMenu system call, 172
Apple event Interprocess Messaging Protocol (AEIMP), 76
Apple event Manager, 77–78
  address descriptor record (AEAddressDesc), 79
  coercing data to other data types, 81
  data structures, 77–80
  descriptor list (AEDescList), 79–80
  descriptor record (AEDesc), 78–79, 81–82
  keyword-specified descriptor record (AEKeyDesc), 79
  routines, 81
Apple events, 5–6, 73, 76–84
  attributes, 77
  client applications, 77
  compatibility, 77
  controlling user interaction preferences, 83
  descriptor types, 78–79
  dispatch tables, 80–81
  Edition Manager, 99–100
  extracting data, 81–82
  keywords, 79
  object model, 341
  parameters, 77–78
  pointers to handlers, 80
  print document(s), 5
  processing, 80
  records (AERecord), 80
  required, 76
  routines, 83
  sending, 82–84
  server applications, 77
  sources, 77
  suspending and resuming processing, 80–81
  targets, 77
  vs. PPC Toolbox, 65
Apple menu, 4, 17–18
Apple Menu Items folder, 4, 18, 22, 322
Apple Programmer's and Developer's Association (APDA), 337
Apple Sound Chip (ASC), 194–195
Apple events, 60
AppleScript, 84, 341
AppleShare, 24
AppleTalk, 9–10, 212, 245, 247
  protocols and drivers, 246–248
AppleTalk Data Stream Protocol (ADSP), 10, 245
AppleTalk Phase II, 245, 248–257
  AppleTalk Transition Queue, 250–252
  application developer features, 249
  compatibility, 253
  transitions, 251
Application menu, 4, 20–21
application-died events, 72
Applications
  32-bit cleanliness, 11
  7.0-compatible, 59–60
  7.0-dependent, 59–60
  7.0-friendly, 59–60
  A/UX and, 59–61
  background notifying user, 12
  client, 77
  controlling colors/gray scales, 270
  exchanging messages, 73
  external code as part of installation, 41
  Finder information available to, 13
  help messages, 171
  idle and sleep state, 13
  installation disk order, 48
  installing, 5, 39
  launching, 298–299
  list, 4, 21
  major switch, 296–297
  memory partition allocation, 75–76
  minor switch, 296–297
  moving, 72
  network installation, 5
  notification request, 301
  preferring outline to bitmap fonts, 116
  receiving session requests, 68–69
  server, 77
  sharing data, 6
  signatures, 13
  simulating forms, 27
  sound channels and commands, 192
  supporting Edition Manager, 93
  transferring data, 63, 87
ASP (AppleTalk Session Protocol), 248

Assertions, 44
AssociateSection routine, 96
ATEvent routine, 251–252
Atoms, 40–41
ATP (AppleTalk Transaction Protocol), 248–250
ATP driver, 247
ATPGetRequest system call, 250
ATPKillAllGetReq system call, 250
ATPreFlightEvent routine, 251
Attention channel, 214
Audible notification, 301
Audio Interchange File Format (AIFF), 189
Audit atoms, 41, 46–47
Authentication, 66
Autokey events, 230

B

Backing store file, 282
BalloonWriter, 171–172
Binary files, 39
BitMaps, 6–7, 32, 104, 107, 264
  color, 262
  converting to QuickDraw, 267
  masking, 267
  offscreen, 274–275
BitMapToRegion system call, 267
Black-and-white icons, 4, 30–31
Blocks, 64, 69
Boot blocks, 41
  atoms, 41
  changes to be made, 47
  parameters, 41
Bottleneck procedures (QuickDraw), 261
buffercmd sound command, 199, 201

C

CalcColorTable routine, 279
Calendars, international, 139–140
CALL statement, 153
CallEditionOpenerProc system call, 99
CallFormatIOProc system call, 99
Cancel button, mapping Escape and Command-period keys to, 35
Canonical records, 314–315
'cbnd' bundle resource, 221, 224
CDEF (control definition procedures), 241, 290–291
'cdef' code resource, 220–222
cGrafPort record, 274
Character codes mapping to key codes, 141
Characters
  closest to position, 126
  converting to uppercase, 136
  coordinates, 127
  end-of-line, 126
  processing incoming, 127
  TrueType, 110–111
  two-byte, 7
checkAllAssertions clause, 43–44
checkAnyAssertions clause, 43
checkAnyAuditRecord clause, 43, 46
checkATVersion clause, 43
checkAuditRecord clause, 43, 46
checkFileContainsRsrcByName clause, 42
checkFileCountryCode clause, 43
checkFileDataForkExists clause, 42
checkFileVersion clause, 43
checkGestalt clause, 42
checkMinMemory clause, 42
checkMoreThanOneAssertion clause, 43–44
checkTgtVolSize clause, 43
checkUserFunction clause, 43–44
Chooser, 35
CL/1. *See* Data Access Language (DAL)
Clauses, 42
Client applications, 77
Clipboard, 87–89
  vs. Edition Manager, 6, 89
'cloc' code resource, 221, 224
Close Selected Icon (Enter) key, 27
CLOSE TABLE statement, 155
CloseEdition routine, 97–98
ClosePicture system call, 267
CMAbort system call, 217
CMAccept system call, 217
CMActivate routine, 218
CMAddSearch routine, 218
CMBreak routine, 218
CMChoose routine, 216
CMClearSearch routine, 218
CMClose system call, 218
cmCloseMsg message, 222
CMDefault routine, 216, 220
cmDefaultMsg message, 223
cmDisposeMsg message, 221
CMEnglishToIntl routine, 219
CMEvent routine, 218
CMGetCMVersion routine, 213
CMGetConfig routine, 216
CMGetConnEnvirons routine, 217
CMGetProcID routine, 216
CMGetRefCon routine, 219
CMGetToolName routine, 219
CMGetUserData routine, 219
CMGetVersion routine, 219
CMIdle system call, 217
cmInitMsg message, 221
CMIntlToEnglish routine, 219
CMIOKill routine, 218
cmL2English message, 224
cmL2Intl message, 224
CMListen system call, 217
CMMenu routine, 218
cmMgetMsg message, 223
cmMsetMsg message, 224
CMNew routine, 216
cmNotSupported message, 223
CMOpen routine, 217
cmOpenMsg message, 221
CMRead routine, 218
cmReadMsg message, 222
CMRemoveSearch routine, 218
CMReset routine, 218
CMResume routine, 219
cmScleanupMsg message, 223
CMSetConfig routine, 216
CMSetRefCon routine, 219
CMSetUserData routine, 219
cmSfilterMsg message, 223
cmSitemMsg message, 223
cmSpreflightMsg message, 223
CMStatus system call, 217
CMValidate system call, 216, 220, 224
cmValidateMsg message, 223
CMWrite routine, 218
CMYK (Cyan-Magenta-Yellow-Black), 10, 269

Color, 10, 29–31
  animating, 271
  bitmaps, 262
  courteous, 270
  inhibited, 271
  palettes, 10
  selecting, 269
  tolerant, 270
Color control panel, 30
Color icons, 4, 30–31, 141
Color Lookup Table (CLUT), 263, 270–271, 273
Color Picker dialog box, 269
Color Picker Package, 10, 269–270
  converting between color models, 270
  CMYK (Cyan-Magenta-Yellow-Black), 10, 269
  HLS (Hue-Lightness-Saturation), 10, 269
  HSV (Hue-Saturation-Value), 10, 269
  RGB (Red-Green-Blue), 10, 269
Comments, 45
COMMIT statements, 155
Communications, 209–210
  devices queue, 240–241
Communications Folder, 22, 216, 228, 323
Communications Resource Manager, 9, 210, 213, 215
  programming, 240–241
Communications Toolbox, 9, 209–242
  architecture, 209–211
  Communications Resource Manager, 9, 210, 213, 215, 240
  Communications Toolbox Utilities, 9, 210–211, 215, 241–242
  compatibility, 212–213
  Connection Manager, 9, 210, 213–219
  data structures, 211–212
  File Transfer Manager, 9, 210, 213, 234–240
  manager
    independence, 210–211
    versions, 213
  network and communications software, 212
  Terminal Manager, 9, 210, 213, 224–234
Communications Toolbox Utilities, 9, 210–211, 215
  programming, 241–242
Comp3to1 routine, 201
Comp6to1 routine, 201
Conditional
  clauses, 42
  sleep demands, 328
Configuration records, 212
Connection Control Block (CCB), 253
Connection Listener (CL), 254, 256–257
Connection Manager, 9, 210, 213–219
  attention channel, 213–214
  connections, 217–218
  code resources, 221
  control channel, 213–214
  data channel, 213–214
  environment, 217
  handling events with routines, 218–219
  interfacing with scripting language, 216
  international support, 219
  programming, 213–219
  record, 215
  routines, 215–219
  tools, 214
Connection records, 211, 215–216, 223
Connection tools, 214
  choosing, 216
  interfacing with scripting language, 223–224
  international translations, 224
  procID, 216
  resources, 224
  setting connection record to default values, 216
  writing, 219–224
Control channel, 214
Control panels, 22
  Color, 30–31
  Keyboard, 141
  Memory, 27–28
  Sharing Setup, 24, 64–65
  Sound, 204
  Users and Groups, 25, 66, 258
Control Panels folder, 22, 322
convertCmd sound command, 201
Cooperative multitasking, 11, 296
Copy command, 89
CopyBits system call, 10, 267
CopyDeepMask routine, 267
CopyMask system call, 267
CopyPalette system call, 273
Correction tools, 22
CountAppFiles routine, 76
CountDITL system call, 242
Courteous colors, 270
Create Publisher... command, 32, 87–89, 92
CreateEditionContainerFile routine, 96
CRMGet1IndResource routine, 241
CRMGet1NamedResource routine, 241
CRMGet1Resource routine, 241
CRMGetCRMVersion routine, 213
CRMGetIndex routine, 241
CRMGetIndResource routine, 241
CRMGetIndToolName routine, 241
CRMGetNamedResource routine, 241
CRMGetResource routine, 241
CRMInstall system call, 240
CRMLocalToRealID routine, 241
CRMRealToLocalID routine, 241
CRMReleaseResource routine, 241
CRMRemove system call, 240
CRMSearch routine, 241
'cscr' code resource, 221, 223–224
'cset' code resource, 220, 223
CTab2Palette system call, 272
CTabChanged routine, 269
CTBGetCTBVersion routine, 213
Currency formats, 139
Cursor, adjusting shape, 72
Custom Install dialog box, 44
Custom Install script, 39, 41
Custom menu definition procedure (MDEF), 171, 182

Custom menus, help resources, 182
CustomGetFile routine, 316, 318–319
CustomPutFile routine, 316, 319
'cval' code resource, 220, 223
Cyan-Magenta-Yellow-Black (CMYK), 10, 269

D

DAs. *See* desk accessories (DAs)
Data
  extracting from Apple events, 81–82
  sharing between applications, 6
Data Access Language (DAL), 7–8, 145–160
  comments, 153
  compound statements, 153
  CURSOR data type, 153
  data manipulation, 152
  data types, 154
  databases, 154–159
  error codes, 156
  examples, 157–158
  expressions, 154
  external variables, 153
  functions, 153
  GENERIC data type, 153
  identifiers, 153
  literals, 154
  local variables, 153
  OBJNAME data type, 153
  operators, 155
  output, 153
  procedures, 153
  program control, 152–153
  running, 150–151
  security, 156–157
  SQL (Structured Query Language), 8, 151
  statements, 152–153
  system variables, 153
  vs. data management system (DBMS), 158–159
Data Access Manager (DAM), 2, 7–8, 32, 145–150, 160–167
  architecture, 148–150
  compatibility, 161
  converting data to text strings, 164–165
  high-level calls, 161–163
  human interface guidelines, 166–167
  initializing, 163
  low-level calls, 161–166
  queries, 31, 150
  query documents, 161–162
  result handlers, 164–165
  retrieving data, 164
  security, 156–157
  sending data to host, 164
Data channel, 213–214
Data fork, xxx
Data structures, 332
  ADSP (AppleTalk Data Stream Protocol), 253
  Apple event Manager, 77–80
  Communications Toolbox, 211–212
  GDevice record, 274–277
  Graphics Device Manager, 274
  GWorld record, 274–275
  PixMaps, 264
  QuickDraw editing, 268–269
  Sound Manager, 189
  SPB (Sound-input Parameter Block), 206–207
Database management system (DBMS), 146–147, 150–151
  errors, 164
  information about, 165–166
  nonrelational, 155
  relational, 155
  queries, 164–165
  session, 163
  vs. Data Access Language (DAL), 158–159
Databases, 146
  Data Access Language (DAL) and, 154–159
  generic application programming interface, 148
  listing, 155
  querying, 148–150
Date formats, 139
DB2, 8
DBBreak system call, 165
DBDisposeQuery system call, 147, 163
DBEnd system call, 147, 163
DBExec system call, 164
DBGetConnInfo system call, 165–166
DBGetError system call, 164
DBGetItem system call, 164–165
DBGetNewQuery system call, 162
DBGetQueryResults system call, 147, 163
DBGetResultHandler system call, 165
DBGetSessionNum system call, 166
DBInit system call, 163
DBInstallResultHandler system call, 165
DBKill system call, 165
DBNewQuery system call, 146
DBRemoveResultHandler system call, 165
DBResultsToText system call, 147, 163–165
DBSend system call, 164
DBSendItem system call, 164
DBStartQuery system call, 146, 163
DBState system call, 164
DBUnGetItem system call, 164
Debuggers, 72
DeferUserFn function, 285
DeleteEditionContainerFile routine, 96–97
DeleteUserIdentity system call, 69
Demand paging memory, 282
DeRez tool, 39
DESCRIBE statements, 155
Descriptor list (AEDescList), 79–80
Descriptor record (AEDesc), 78–79
Desk accessories (DAs), 4
  Find File, 18
  launching, 299
  listing active, 21
  locations to keep, 23
Desktop
  listing objects, 18
  moving around on, 26–27
  objects, 26–27
  stationery pad, 27
  Trash, 27

Desktop database, 319–321
  application connecting
    information, 320
  file comment calls, 320–321
  icon bitmap calls, 320
  system calls, 320–321
Desktop Folder, 22, 322
despWrite system call, 255
Devanagari Script System, 128
Device drivers, 59, 328
DeviceLoop routine, 276
Devices, 262–263
Diacritical characters, stripping out,
  136
Dialog boxes, 33–35, 123, 318–319
  buttons, 35–36
  complex, 334
  help messages, 176–177
  help states, 173
  keyboard navigation, 4, 33–35
  managing, 242
  modal, 34
  modeless, 34
  movable modal, 4, 33–34
Direct color
  devices, 262
  PICT2 format, 265–266
  PixMaps, 264–265
Direct memory access (DMA), 340
DirectBitsRect opcode, 265–266
DirectBitsRgn opcode, 265–266
Directory IDs, 309
Disks, order during installation, 48
Dispatch tables, 80–81
Display, TextEdit synchronizing
  with keyboard, 124
DisposeColorPickMethod routine,
  279
DisposeHandle system call, 287
DisposePalette system call, 271
DisposePictInfo routine, 278–279
DisposeResource system call, 271
DisposGDevice routine, 277
DisposGWorld system call, 275
DisposHandle routine, 288
DitherCopy system call, 267
'DITL' resource, 176–177
'DLOG' resource, 318
Document string resources,
  321–322
Documents
  saving
    with publishers, 89
    with sections, 94
  subscribing, 89
  utilizing editions, 89
doFace mode, 127
doToggle mode, 127
Double page fault, 285
DragGrayRegion system call, 267
DrawPicture system call, 267
DrawText routine, 117
DSP driver, 247, 253–256
dspAttention system call, 255–257
dspCLDeny system call, 254
dspCLInit system call, 254, 256
dspCLListen system call, 254,
  256–257
dspClose system call, 254, 256–257
dspCLRemove system call, 254,
  257
dspDeny system call, 257
dspInit system call, 254, 256–257
dspOpen system call, 254, 256–257
dspOptions system call, 255–256
dspRead system call, 255–257
dspRemove system call, 254,
  256–257
dspReset system call, 255
dspStatus system call, 254
dspWrite system call, 256–257

E

Easy Install
  rules, 42, 44
  script, 39
Edit menu, 18, 32, 34, 89, 92, 181
Edition files, 6, 31, 90, 94
  deleting, 96–97
  file types, 96
  information about, 98
  locating, 89, 91
  moving to another volume, 92
  updating, 89
Edition icons, 31
Edition Manager, 5–6, 32, 73,
  87–101, 314
  Apple events, 99–100
  compatibility, 95
  format marks, 98–99
  high-level events, 100
  implementing support, 100
  reading and writing edition data,
    96
  section records, 95–96
  sections, 96
  starting, 95
  subscribing to files, 99
  supporting from applications, 93
  updating alias record, 96
  vs. Clipboard, 6, 89
Edition opener procedures, 99
EditionHasFormat routine, 98
Editions, 92
  reading/writing data, 97–98
  utilizing in documents, 89
EmptyHandle routine, 288
Emulation tools, 22
End-of-line characters, 126
ENET driver, 247
Enter key, 35
Environmental selectors, 52–54
Environs system call, 51
EtherTalk, 247
Event Manager, 74
EventAvail system call, 72–73, 296
Events, xxxii-xxxiii, 5–6
  Apple, 5–6, 73, 76–84
  application-died, 72
  autokey, 230
  Connection Manager routines,
    218–219
  File Transfer Manager routines,
    237
  high-level, 71, 73–76, 84
  key-down, 230
  low-level, 71–72
  mouse-down, 230
  mouse-moved, 72
  operating-system, 72
  resume, 72
  suspend, 72
  Terminal Manager routines, 230
EXECUTE statements, 155
Exp1to3 routine, 201
Exp1to6 routine, 201
Extensions folder, 22, 322

F

Farallon Sound Recorder, 188

'fbnd' bundle resource, 238
'fdef' code resource, 238–240
  messages, 239
FETCH statement, 155
File atoms, 40–41, 45
File IDs, 12, 308–310
File Manager
  compatibility, 308, 313
  file IDs, 308–310
  files, 309–311
  high-level HFS calls, 312
  routines, 311–313
File menu, 18, 25, 32, 166
File servers, 24–25, 245–246, 257
File Sharing, 9–10, 24–25, 245–246, 257–259
File system, 12–13, 307–323
File Transfer Manager, 9, 210, 213, 217
  file transfer records, 211, 235–236
  international support, 238
  programming, 234–240
  routines, 236–238
File transfer records, 211, 235–237
File transfer tools, 9, 22, 234–235, 238–240
Files
  .r, 39
  alias, 308
  backing store, 282
  binary, 39
  control panel, 22
  copying or deleting, 41
  edition, 6, 31
  generic reference, 13
  grouping by function, 21–22
  keeping track of, 308–310
  labeling, 19–20
  locating, 4, 18
  opening from aliases, 23
  pathname, 47
  pointing to original file, 23
  preferences, 22
  query document, 31
  recording sounds to, 205
  search criteria, 12–13
  searching for, 310–311
  sound, 189
  specifying, 310
  stationery, 31
  subscribing to, 99
  suitcase, 22
  swapping data and system fork, 309
  System, 137
  SysTypes.r, 138
  temporary, 22
  transferring, 236–237
  viewing by label, 19
Find Again command, 18
Find File desk accessory (DA), 18
Find... command, 4, 18
Finder, 4, 17–27, 314
  document string resources, 321–322
  help message, 179
  information, 13, 319–320
  menu bar, 17–18
  new icon types, 321
Finder mode, 11
FindFolder system call, 323
FindScriptRun system call, 140
FindWord procedure, 127, 140
Fix2SmallFract routine, 270
Fixed-frequency scheduling, 304
'floc' code resource, 238
flushCmd sound command, 200
FlushFonts system call, 116
fmAbortMsg message, 240
FMDefault routine, 238
fmDisposeMsg message, 240
fmExecMsg message, 240
fmInitMsg message, 240
fmStartMsg message, 240
'fmts' data type, 94
FMValidate routine, 238
'FOND' resources, 46, 107–109, 132, 139
Font Manager, 60, 105
  compatibility, 116
  flushing caches, 116
  measurements on string of characters, 116–117
  not scaling font to fit line, 117
  selecting fonts, 115
  TrueType, 116–117
Font numbers, 104
'FONT' resources, 46, 104, 107–108, 266
Font/DA Mover, 4, 22–23, 105
FontName opcode, 266
Fonts, 103–118
  advance-width character measurement, 117
  bitmap, 6–7, 32, 104, 107
  family, 107
  headers, 109
  help balloons, 183
  history, 103–106
  identifying by name, 106–108
  inconsistent information, 46
  installing, 4, 22–23
  left-side bearing character measurement, 117
  points, 104
  PostScript, 105–108, 110
  selecting with Font Manager, 115
  sizes, 32, 60, 117
  TrueType, 5–7, 108–116
FOR EACH statement, 155
Format marks, 98–99
Format2Str routine, 141
FormatStr2X routine, 141
FormatXString routine, 141
freeCmd command, 193
'fscr' code resource, 238
'fset' code resource, 238
FSOpen routine, 312
FSpCreateResFile routine, 313
FSpExchangeFiles system call, 309
FSpOpenResFile routine, 313
FSSpec records, 12, 310, 318
FTAbort routine, 237
FTActivate routine, 237
FTChoose routine, 236
FTDefault system call, 237
FTDispose routine, 237
FTEnglishToIntl routine, 238
FTEvent routine, 237
FTExec routine, 237
FTGetConfig routine, 236
FTGetFTVersion routine, 213
FTGetProcID system call, 237
FTGetRefCon routine, 238
FTGetToolName routine, 238
FTGetUserData routine, 238
FTGetVersion routine, 238
FTIntlToEnglish routine, 238
FTMenu routine, 237

FTNew routine, 236
FTResume routine, 237
FTSetConfig routine, 236
FTSetRefCon routine, 238
FTSetUserData routine, 238
FTStart routine, 237
FTValidate routine, 237
FUnits, 114
'fval' code resource, 238

G

GDEF resource selector functions, 56
GDevice record, 274–277
GDeviceChanged routine, 269
Generative clauses, 42
Generic file reference, 13
Gestalt Manager, 51–56, 59
Gestalt system call, 51–55, 65, 77, 95, 123, 136, 202, 212
 selectors, 52–54
 result types, 54–55
Get Data... command, 166
Get Info dialog box, 27
Get Info... command, 27
GetAliasInfo system call, 316
GetAppFiles routine, 76
GetColor routine, 269
GetCTable system call, 268
GetCurrentProcess system call, 299–300
GetCVariant system call, 291
GetDefaultUser system call, 69
GetDeviceList routine, 276
GetEditionFormatMark system call, 96, 99
GetEditionInfo routine, 98
GetEditionOpenerProc system call, 99
GetEntryColor system call, 271
GetEntryUsage system call, 271
GetEnvirons routine, 136
GetFrontProcess system call, 300
GetGDevice routine, 276
GetGray system call, 267
GetGWorld system call, 275
GetGWorldDevice system call, 275
GetHandleSize routine, 288
GetLastEditionContainerUser routine, 97
GetLocalZones system call, 250
GetMainDevice routine, 276
GetMaxDevice routine, 276
GetMaxZone system call, 250
GetNewPalette system call, 271
GetNextDevice routine, 276
GetNextEvent system call, 59–60, 72–73, 286–287, 296
GetNextProcess system call, 300
GetOutlinePreferred system call, 116
GetPalette system call, 272
GetPaletteUpdates system call, 273
GetPhysical system call, 285
GetPictInfo routines, 277, 279
GetPixBaseAddr system call, 275
GetPixelsState system call, 275
GetPixMapInfo routine, 278–279
GetPortNameFromProcessSerialNumber routine, 75
GetPreserveGlyph system call, 117
GetProcessInformation system call, 300
GetProcessSerialNumberFromPortName routine, 75
GetResource system call, 271
GetScript routine, 136
GetSpecificHighLevelEvent system call, 74
GetWVariant system call, 291
GetZoneList system call, 250
Global variables
 low-memory, 59
 scripts, 136
GotoPublisherSection routine, 98–99
Graphical human interface, 2
Graphics Device Manager, 10, 274–277
 data structures, 274–277
Graphics devices, 10
 communicating, 274–277
Grid-fitting, 114
GWorld record, 274–275

H

Handles, xxix
HandleZone routine, 288
Hard disks, space and virtual memory (VM), 15
Hardware
 device drivers, 59
 managers, 59, 325–329
 sounds, 195
HasDepth routine, 277
HClrRBit routine, 288
HCreateResFile routine, 313
'hdlg' resource, 176–177, 181–184
Head table in sfnt resources, 109
HeaderOp opcode, 266–267
Heap, xxviii-xxix
Help, turning on/off, 21
Help balloon, 169, 171
 displaying True/False, 183
 fonts, 183
Help command, 32
Help Manager, 8, 21, 32, 169–185
 compatibility, 172
 custom menus, 182
 flags, 174
 help messages, 173–174
 hot rectangles, 172–173
 mapping from window titles to help resource, 178–179
 modal dialog boxes, 181
 modeless dialog boxes, 181–182
 movable window objects, 183
 resources, 174–180
 routines, 183–184
 states, 173
 windows, 181–182
Help menu, 21, 32, 34, 169, 181
Help messages, 171, 173–174
 alerts and dialog boxes, 176–177
 finder, 179
 international support, 172
 NIL string, 176
 overriding default, 179–180
 set of window objects, 177–178
 standard menus, 175–176
 windows, 176–177
 writing, 185–186
Help resources for menus, 184–185
'hfdr' resource, 179
HGetState routine, 288
Hierarchical File System (HFS), 307
 equivalent routines for MFS routines, 312
 high-level calls, 312

High-level events, 5–6, 71, 73–76, 84
additional information about, 74
Apple, 5
calling manager routines, 74
describing, 73–74
Edition Manager, 100
mask set in event record, 73
message field, 73–74
retrieving, 74
sending to another application, 74–75
Historical installation information, 41
Hlock routine, 288
HLS (Hue-Lightness-Saturation), 10, 269
HMCompareItem statement, 180
HMGetBalloons routine, 182
HMGetDialogResID routine, 183
HMGetFont routine, 183
HMGetFontSize routine, 183
HMGetHelpMenuHandle routine, 172
HMGetMenuResID routine, 184
HMIsBalloon routine, 183
HMNamedResource statement, 180
'hmnu' resource, 175–176, 180, 183–185
HMRemoveBalloon system call, 182–183
HMScanTemplateItems routine, 184
HMSetBalloons system call, 182
HMSetDialogResID routine, 183
HMSetFont routine, 183
HMSetFontSize routine, 183
HMSetMenuResID routine, 183–184
HMShowBalloon system call, 183
HMShowMenuBalloon system call, 182
HNoPurge routine, 288
HoldMemory system call, 284–285
HOpenDF routine, 312
HOpenResFile routine, 313
HGot rectangles, 172–173, 177–178
Hot regions, 172
'hovr' resource, 179–180
HPurge routine, 288
'hrct' resource, 177–178, 181–182, 184
HSetRBit routine, 288
HSetState routine, 288
HSV (Hue-Saturation-Value), 10, 269
Human interface
color, 29–31
design, 29–31
dialog boxes, 33–35
guidelines, 29–36
Data Access Manager (DAM), 166–167
sound, 203
menus, 32
windows, 33
HUnlock routine, 288
'hwin' resource, 178–179, 181

I

'icmt' (Installer comments) resource, 45, 47
Icons
black-and-white, 4, 30–31
color, 4, 30–31
edition, 31
information about, 22
large, 4
new types, 31, 321
query files, 31
selecting next, 27
small, 4, 31
stationery, 31
ImageWriter, 105
'inaa' (action atom) resource, 46, 48
'inat' (audit atom) resource, 46–47
'inbb' (block boot atom) resource, 47
Indexed color devices, 262–263
'indo' (disk order) resource, 48
'infa' (file atom) resource, 45
Informational selectors, 52, 54
Informix, 8
'infr' (rule framework) resource, 44
'infs' (file specification) resources, 45–47
Ingres, 8
Inheritance, 332
inhibited color, 271
InitCM system call, 215
initCmd command, 193
InitCRM system call, 215, 228, 236, 240–241
InitCTBUtilities system call, 215, 228, 236, 241
InitDBPack system call, 146, 162–163
InitEditionPack system call, 95
InitFT system call, 236
InitPickMethod routine, 279
INITS, 22
InitTM system call, 228
'inpk' (package) resource, 44–45, 47
'inra' (resource atom) resource, 45–46
'inrl' (rule) resource, 42, 44
Installer, 5, 37–49
AppleShare server installation, 37
assertions, 44
comments, 47
Custom Install script, 39
customizing, 48
Easy Install script, 39, 42, 44
expert mode, 37
live installation, 37
one-button mode, 37
Resource Manager, 39
resources, 46–47
scripts, 5, 37, 39–41, 47
splash screen, 48
version 3.2, 37
Instance variables, 332
InsTime system call, 305–306
InsXTime system call, 305–306
Integrated Voice Data Manager, 212
Interapplication communications (IAC), 5–6
low-level, 63
Interfaces
guidelines, 4–5
programmable to NuBus cards, 13
standard user, 9
International support, 121–122
calendars, 132
Connection Manager, 219
connection tools, 224
File Transfer Manager, 238

help messages, 172
improvements, 136–142
punctuation, 127
resources, 137–142
script or writing system, 128–132
Terminal Manager, 231
text buffering, 125–126
TextEdit, 123–124
writing international software, 142–143
International Utilities Package, 122, 128, 136–137
Interpreter, 114
'INTL' resource, 139
IntlTokenize routine, 141
IPCListPorts system call, 67
ISDN (Integrated Services Digital Network) Developer's Toolkit, 212
IsOutline system call, 116
IsRegisteredSection routine, 96, 100
'itl0' resource, 137–139
'itl1' resource, 137–140
'itl2' resource, 136–140
'itl4' resource, 137–141
'itlb' resource, 138, 141
'itlc' resource, 138–139
'itlk' resource, 142
'itlm' resource, 137, 139
IUClearCache routine, 137, 140
IUCompPString routine, 137
IUCompString routine, 137
IUDateString routine, 137
IUEqualPString routine, 137
IUEqualString routine, 137
IUGetIntl routine, 137
IUGetItlTable routine, 137
IULangOrder routine, 137
IULDateString routine, 137
IULTimeString routine, 137
IUMagIDPString routine, 137
IUMagIDString routine, 137
IUMagPString routine, 137
IUMagString routine, 137
IUScriptOrder routine, 137
IUSetIntl routine, 137
IUStringOrder routine, 137
IUTextOrder routine, 137
IUTimeString routine, 137

J

Justification modes, 127–128

K

KCAP resource, 141–142
KCHR resource, 138, 141
'kcs#' resource, 138, 141
'kcs4' resource, 141
'kcs8' resource, 141
Key codes, mapping to character codes, 141
Key-down events, 230
Keyboard codes, 141–142
Keyboard control panel, 141
Keyboard menu, 141
Keyboards
color icons, 141
moving around on desktop, 26–27
navigating in dialog boxes, 4
physical layout, 141–142
TextEdit synchronizing with display, 124
KeyTrans routine, 142
Keyword-specified descriptor record (AEKeyDesc), 79
'KSWP' resource, 142

L

Label menu, 17, 19–20
Languages
codes sorting order, 139
international support, 128–132
LAP (Link Access Protocol) Manager, 250–251
LAPAddATQ system call, 251
LAPRmvATQ system call, 251
Large icons, 4
LaserWriter, 105–106
LaunchApplication routine, 298–299
LaunchDeskAccessory routine, 299
lineJustify opcode, 266
loadCmd sound command, 201
Local variables, scripts, 136
Locality of reference, 285–286
LocalTalk, 247
LockMemory system call, 285
LockMemoryContiguous system call, 285
LockPixels system call, 275
Low-level
events, 71–72
interapplication communications (IAC), 63
Low-memory globals, 59
LowerText routine, 140

M

MacApp 2.0, 334–337
MacBrowse, 334
memory, 335–336
programmer resources, 337
ViewEdit tool, 334
MacApp Developer's Association (MADA), 337
MacBrowse, 334
MACE (Macintosh Audio Compression and Expansion) synthesizers, 194–195
MACEVersion system call, 202
Macintosh
IIsi, digital sound, 188
LC, digital sound, 188
operating system, xxvii
Portable, 327–329
Macintosh Programmer's Workshop (MPW), 184
Mark menu, 184–185
Marking selections, 184
MatchAlias system call, 315–316
mcWriteMsg message, 222
MDEFs (Menu DEFinition procedures), 8, 182
Memory
addresses for 32-bit cleanliness, 291–292
deallocating local, 221
demand paging, 282
MacApp 2.0, 335–336
object-oriented programming (OOP), 335–336
pages, 15, 282
partition allocation for application, 75–76
protection, 341
requirements, 14
temporary, 11, 286–288

Memory control panel, 27–28
Memory management unit (MMU), 14, 282
Memory Manager, 11, 281–292, 314
  32-bit cleanliness, 289–292
  backing store file, 282
  temporary memory, 286–288
  virtual memory (VM), 281–286
Menu items, help states, 173
MENU resource, 184
Menus, 32
  help
    messages, 175–176
    resources, 180, 184–185
    states, 173
  new standard commands, 32
  pop-up, 32
MenuSelect system call, 180
Methods, 332
MFFreeMem routine, 288
MFMaxMem routine, 288
MFTempDisposHandle routine, 287–288
MFTempHLock routine, 288
MFTempNewHandle routine, 288
MFTopMem routine, 288
Microsoft Windows, 1–2
Modal dialog boxes, 34
  feedback, 35–36
  help resources, 181
Modeless dialog boxes, 34
  help resources, 181–182
Modifiers, 193
  communications with Sound Manager, 193
  installing, 203
Monitor cdev, 262
Motorola 68851 PMMU chip, 14
Mouse-down events, 230
Mouse-moved events, 72
Movable modal dialog boxes, 4, 33–34
MPP driver, 247, 250–251, 255–256
  AppleTalk Transition Queue, 249
  current node information, 252
MPPOpen system call, 255–256
MultiFinder, 20, 59–60, 72, 75, 295
Multitasking
  cooperative, 11, 296
  preemptive, 295, 341–342

N

NBP (Name Binding Protocol), 249
  metacharacters, 67
  wildcard characters, 252–253
NBPLookup system call, 256
NBPRegister system call, 256
NChar2Pixel routine, 135
NDrawJust routine, 135
Networks
  booting, 342
  edition files, 94
  installing applications, 5
New print architecture, 340
NewAlias system call, 315
NewAliasMinimal system call, 315–316
NewAliasMinimalFromFullpath system call, 315–316
NewControl system call, 241
NewGDevice routine, 276
NewGestalt system call, 55
NewGWorld system call, 274
NewPalette system call, 271
NewPictInfo system call, 278
NewPtr system call, 256
NewPublisherDialog routine, 97
NewPublisherExpDialog routine, 97
NewSection routine, 95–96
NewSubscriberDialog routine, 97–98
NewSubscriberExpDialog routine, 98
NFindWord routine, 135
'NFNT' resources, 46, 107–109, 266
NIL string, 176
NMeasureJust routine, 135
NMInstall system call, 302
NMRemove system call, 302
Non-Macintosh file support, 340
NoPurgePixels system call, 275
Notification Manager, 12, 166, 191, 295, 301–302, 304
  alert notification, 302
  audible notification, 301
  notification
    request, 301
    response procedure, 302
  polite notification, 301
NPixel2Char routine, 135
NPortionText routine, 136
NSetPalette system call, 273
nTEWidthHook procedure, 126–127
NuBus cards, 325
  programmable interface, 13
NuLookup routine, 242
Numbers, formats, 139
NuPLookup routine, 242

O

Object-based programming, 334
Object-oriented programming (OOP), 331–337, 343
  inheritance, 332
  instance variables, 332
  memory, 335–336
  methods, 332
  objects, 332–333
  performance, 335–336
Objects, 332–333
ocAccept mode, 254
ocEstablish mode, 254
ocPassive mode, 254
ocRequest mode, 254
On Location, 18
Open Query command, 32
OPEN TABLE statement, 155
Open... command, 35
OpenCPicture system call, 267
OpenCPort system call, 275
OpenDeskAcc system call, 299
OpenDF routine, 312
OpenDriver system call, 253, 255–256
OpenEdition routine, 98
OpenNewEdition routine, 97
OpenPicture system call, 267
Operating system, xxvii-xxviii, 2
Operating System Event Manager, 71
Operating-system events, 72
Oracle, 8
OutlineMetrics system call, 116–117
Output devices, graphics, 10

P

Packages, 44
  atoms, 41
  listings in Custom Install dialog box, 44
  managing, 41
  removable, 44
  resource type and ID of components, 45
Page fault, 282
Pages, 282
  supporting network and communications protocols, 212
Paging device, 282
Palette Manager, 10, 270–273
  animating color, 271
  courteous colors, 270
  inhibited color, 271
  palettes, 271–273
  tolerant colors, 270
Palette2CTab system call, 272
Palettes, 272–273
Parallel routines, 137
Parameter packet, 66
Paste command, 89
PATalkClosePrep function, 251
Pathnames, 47
pauseCmd sound command, 200
PBCatMove routine, 310
PBCatSearch routine, 12–13, 310–311
PBCreateFileIDRef system call, 309
PBDeleteFileIDRef system call, 309
PBDTAddAPPL system call, 320
PBDTAddIcon system call, 320
PBDTCloseDown system call, 320
PBDTFlush system call, 321
PBDTGetAPPL system call, 320
PBDTGetComment system call, 320
PBDTGetIcon system call, 320
PBDTGetPath system call, 320
PBDTOpenInform system call, 320
PBDTRestComment system call, 321
PBDTRmvAPPL system call, 320
PBDTSetComment system call, 320
PBExchangeFiles system call, 309
PBGetCatInfo routine, 309–310
PBGetIconInfo system call, 320
PBGetVolMountInfo system call, 313
PBGetVolMountInfoSize system call, 313
PBHDelete routine, 310
PBHGetVolParms system call, 308, 311
PBHOpenDF system call, 312
PBHRename routine, 310
PBOpenDF system call, 312
PBSetForeignPrivs system call, 311
PBVolumeMount system call, 313
PGGetAppleTalkInfo system call, 252
PGGetForeignPrivs system call, 311
PICT file format, 10, 94, 265, 277–278
'PICT' resource, 48, 173
PICT2 format, 267
  direct color, 265–266
Picture Utilities Package, 10, 277–279
PixMap records, 277–279
PixMap32Bit system call, 275
PixMaps, 10, 262, 267
  converting to QuickDraw, 267
  data structures, 264
  direct color, 263–265
  locking down, 275
  masking, 267
PixPatChanged routine, 269
Playback synthesizers, 193
'pltt' resources, 271–272
PmBackColor system call, 273
PmForeColor system call, 273
PMMU chip. *See* Motorola 68851 PMMU chip
Points, 104
Polite notification, 301
Pop-up menus, 32
PortChanged routine, 269
Ports, 64
  changing, 68
  converting names to and from serial numbers, 75
  filter function, 67
  listing, 67
  location, 64
  naming, 64
  opening, 67
  shutting down, 69
PostHighLevel system call, 74
PostScript fonts, 105–108, 110
PostScript language, 151
Power Manager, 13, 325, 327–329
  conditional sleep demands, 328
  sleep requests, 328
  unconditional sleep demands, 328
PPC Toolbox, 63–71, 84
  authentication of remote user, 66
  calling, 66–71
  compatibility, 65
  completion routine, 66
  example, 69–71
  managing services, 65–66
  ports, 64, 67–69
  receiving and sending data, 69
  terminology, 64
  user authentication, 69
  vs. Apple events, 65
PPC Toolbox. *See* Interapplication communications
PPCAccept system call, 69–70
PPCBrowser system call, 67, 70
PPCCall system call, 71
PPCClose system call, 69, 71
PPCEnd system call, 69, 71
PPCInform system call, 66, 68–69
PPCInit system call, 66
PPCOpen system call, 67, 69
PPCRead system call, 66, 69–70
PPCReject system call, 69
PPCStart system call, 68
PPCWrite system call, 66, 69–70
Preemptive multitasking, 295, 341–342
Preferences files, 22
Preferences folder, 22, 322–323
PrimeTime routine, 304–306
print document(s) Apple event, 5
Print Monitor, 301
Printer drivers, 22, 106

PrintMonitor Documents folder, 22, 322
PROCEDURE statement, 153
Procedures, 332
Process Manager, 11, 295–301
applications, 298–299
compatibility, 301
desk accessories (DA), 299
process scheduling and switching, 296–297
process serial numbers (PSN), 12, 296
Processes, 11–12, 295–306
background notifying user, 301–302
information about other, 299–301
listing active, 20–21
Program-to-Program Communications (PPC), 63
Programs
connecting, 64
controlling with Data Access Language (DAL), 152–153
disconnecting, 64
receiving and sending data, 69
supporting connections, 64
Protocol handling, 221
'prvw' data type, 94
PSendResponse system call, 250
PSMakeFSSpec system call, 310
Publisher Options... dialog box, 90, 97
Publisher/Subscriber Options... command, 32, 87, 92
Publishers, 88–98
alias record, 94
closing, 97–98
dialog boxes, 97
displaying, 92
opening document, 98
saving documents, 89
Punctuation, international support, 127

Q

QDDone system call, 267
'qdef' resource, 162
QDError routine, 267
'qrsc' resource, 162
Queries, 150
executing, 164
Query documents, 31, 150, 161–162
icons, 31
resources, 162
Query records, 162, 163
Queue for communications devices, 240–241
QuickDraw, xxxi-xxxii, 261–279, 342
See also 32-bit QuickDraw
32-bit, 10, 262–270
bottleneck procedures, 261
Color Picker Package, 269–270
data structures, editing, 268–269
Graphics Device Manager, 274–277
history, 261–262
Palette Manager, 270–273
Picture Utilities Package, 277–279
regions, xxxi
revised, 343
QuickTime, 342
quietCmd sound command, 200
quit Apple event, 5

R

Read system call, 286
ReadEdition routine, 98
ReadPartialResource routine, 313
RealFont routine, 117
RecordColors routine, 279
Recording sounds, 203
high-level interface, 205
low-level interface, 206–207
RecordPictInfo routine, 278
RecordPixMapInfo routine, 278–279
Records, 211–212
RecoverHandle, 288
Rescued Items from folder, 323
Region codes, sorting order, 139
RegisterSection routine, 96
Relational databases standard programmatic interface to management systems, 7–8
ReplaceGestalt system call, 55–56
ReplaceText routine, 136
reportError clause, 43
required Apple events, 76
Rescued Items from folder, 22
ResEdit, 48
ResolveAlias system call, 315
ResolveAliasFile routine, 316
Resource alias, 315
Resource atoms, 41, 45–46
Resource fork, xxx
Resource Manager, 313
Installer and, 39
Resources, xxix-xxxi, 5
copying or deleting, 41
installing, 39
international, 137–142
organization in scripts, 40–41
RestoreDeviceClut system call, 273
Result handlers, 164–165
resume events, 72
RetrievePictInfo routine, 278
RetrievePixMapInfo routine, 279
Rez MPW tool, 39–40
RGB (Red-Green-Blue), 10, 262, 269
RmvTime system call, 305–306
ROLLBACK statements, 155
Roman script system, 21, 128, 132
Rules, 41–42, 44

S

SameProcess system call, 301
Sampled sound, 199–200
Save As... command, 35, 309
Save command, 309
SaveBack system call, 273
SaveFore system call, 273
Scaler, 114
Scan converter, 114
Script codes, 139
Script Manager, 121–122, 127, 142–143
compatibility, 136
improvements, 128–136
routines, 133–136
TextEdit compatibility, 123
Script menu, 21
Script systems, 128–132
ScriptCheck MPW tool, 40
Scripts, 39

comparing strings in different, 137
Custom Install, 39, 41
developing, 39–40
Easy Install, 39
files in symbolic manner, 47
global variables, 136
Installer, 5, 37
key combinations for switching, 142
local variables, 136
organization, 40–41
rules, 42
SCSI Manager, 340
Section records, 95–96
SectionOptionsDialog routine, 97–98
SectionOptionsExpDialog routine, 97–98
Sections, 92
information about edition file, 98
saving documents, 94
SELECT statement, 155
Selector functions, 56
Selectors, 52–54
environmental, 52–54
informational, 52, 54
Server applications, 77
Session, 64
requests, 68–69
Set Startup... command, 20
SetA5 routine, 292
SetCurrentA5 routine, 292
SetDepth routine, 277
SetEditionFormatMark routine, 96, 99
SetEditionOpenerProc system call, 99
SetEntryColor routine, 271–272
SetEntryUsage routine, 271–272
SetEnvirons routine, 136
SetGDevice routine, 277
SetGWorld system call, 275
SetHandleSize routine, 288
SetOutlinePreferred system call, 115
SetPalette system call, 272–273
SetPaletteUpdates system call, 273
SetPixelsState system call, 275
SetPreserveGlyph system call, 117
SetResLoad system call, 313
SetResourceSize system call, 313
SetScript routine, 136
SetupAIFFHeader routine, 206–207
SetupSndHeader routine, 206–207
'sfnt' resources, 109–110, 115, 266
Sharing command, 25
Sharing dialog box, 25
Sharing Setup control panel, 24, 64–65
ShortenDITL system call, 242
Show/Hide Borders command, 32, 89, 92–93
'SIZE' resource, 59, 75–76, 126, 292
sizeCmd sound command, 201
Sleep requests, 328
Slot Manager, 13, 325–327
compatibility, 327
initialization process, 326
routines, 327
Slot Resource Table, 326
Small icons, 4, 31
SmallFract2Fix routine, 270
'snd ' resource, 189, 191, 196, 199
SndAddModifier system call, 203
SndChannelStatus system call, 202
SndControl system call, 201
SndDisposeChannel system call, 198, 200
SndDoCommand system call, 198–200
SndDoImmediate system call, 198–200
SndGetSysBeepstate routine, 191
SndManagerStatus system call, 202
SndNewChannel system call, 193, 198–199, 203
SndPauseFilePlay system call, 200
SndPlay system call, 190–191, 201
SndRecord dialog box, 205
SndRecord system call, 205
SndRecordToFile system call, 205
SndSetSysBeepState routine, 191
SndSoundManagerVersion system call, 202
SndStartFilePlay system call, 190–191, 199
SndStopFilePlay system call, 200
'snth' resource, 189, 193–194
Software
installation tools, 48–49
localization, 142
system-wide functionality, 22
Sony chip, 195
Sound channels, 192
capacity, 200
limitations, 195
managing, 198, 200
modifiers, 193
status, 202
Sound control panel, 204
Sound Manager, 9, 187–207
allocating/deallocating sound channel, 191
applications, 192
architecture, 191–195
ASC (Apple Sound Chip), 194
commands, 196–198
compatibility, 201–202
high-level calling, 190–191
installing instruments, 199
low-level calling, 198–201
modifiers, 193, 203
resources, 189
sampled sound from disk, 199–200
sound channel, 198, 200, 202
sound input devices, 204
synthesizers, 188, 193–195
wave tables, 188
soundCmd sound command, 199
Sounds, 187–189, 191
Apple Sound Chip (ASC), 195
commands, 196–198
compression and expansion, 194–195, 201
data structures, 189
digital conversion, 9
digitally recorded, 188–189
files, 189
hardware, 195
human interface guidelines, 203
input, 203–204, 206
installing, 23
recording, 203
Sony chip, 195
wave tables, 188
Source-code browser, 334

Sources, 77
SPB (Sound-input Parameter Block) data structure, 206–207
SPBBytesToMilliSeconds routine, 207
SPBCloseDevice routine, 207
SPBGetDeviceInfo system call, 206
SPBGetIndexedDevice system call, 206–207
SPBGetRecordingStatus system call, 207
SPBMilliSecondsToBytes routine, 207
SPBOpenDevice system call, 206
SPBPauseRecording routine, 206
SPBRecord system call, 206
SPBRecordToFile system call, 206
SPBResumeRecording routine, 206
SPBSignInDevice routine, 207
SPBSignOutDevice routine, 207
SPBStopRecording routine, 206
SPBVersion system call, 202
Special menu, 20
Splash screen, 48
SQL (Structured Query Language) and Data Access Language (DAL), 8, 151
sResource, 326
Standard File Package, 13, 35, 314, 316–319
  compatibility, 319
  customizing, 89
Standard user interface, 9
StandardFileReply data structure, 316, 319
StandardGetFile routine, 316–319
StandardPutFile routine, 316, 318–319
StartSecureSession system call, 68, 70
Startup Items folder, 22, 322
Stationery file, 31
Stationery pad, 27
Stop All Editions... command, 32, 93
STPreFlightEvent system call, 252
'STR ' resource, 173
'STR#' resource, 162, 173
Str2Format routine, 141
Strings
  comparing, 136–137, 140
  date and time, 137
  replacing, 136
StripAddress function, 291
StripText routine, 136, 140
StripUpperText routine, 136, 140
StuffIt, 48
'styl' resource, 173
Subscribe To... command, 32, 87–89, 92
Subscriber Options... command, 89
Subscriber Options... dialog box, 90, 98, 100
Subscribers, 88, 90–92, 97–98
Subscribing, 88–96
Suitcase file, 22
suspend events, 72
syncCmd sound command, 200
Synthesizers, 193–195
  MACE (Macintosh Audio Compression and Expansion), 194–195
  playback, 193
  utility, 193
SysBeep system call, 187, 190–191
SysEnvirons system call, 51
System 7
  aliases, 13
  A/UX, 51, 56–61
  components, 3–13
  file system, 12
  future features, 339
    Apple event object model, 341
    AppleScript, 341
    File System Manager, 340
    Layout Manager, 340
    memory protection, 341
    multimedia support, 342
    multiple address spaces, 341
    network booting, 342
    new print architecture, 340
    object-oriented programming (OOP), 343
    preemptive multitasking, 341–342
    revised QuickDraw, 343
    revised SCSI Manager, 340
  hardware managers, 13
  high-level events, 5–6
  history, 3
  human interface guidelines, 4–5, 29–36
  interapplication communications, 5–6
  international services, 7
  memory requirements, 14
  passwords and user names, 66
  processes, 11–13
  running, 14–15
  strategy behind, 1–2
System crash, 22
System file, 137
System Folder, 4, 13, 21–23, 216, 228, 322–323
  Apple Menu Items folder, 18, 22, 322
  changes, 308
  Communications Folder, 22, 323
  Control Panels folder, 22, 322
  Desktop Folder, 22, 322
  Extensions folder, 22, 322
  Font/DA Mover, 22–23
  Preferences folder, 22, 322
  PrintMonitor Documents folder, 22, 322
  Rescued Items from folder, 22, 323
  Startup Items folder, 22, 322
  System folder, 22, 323
  Temporary Items folder, 22, 322
  Trash folder, 22, 323
SysTypes.r file, 138

T

Targets, 77
'tbnd' bundle resource, 232
'tdef' code resource, 231–234
teCenter justification mode, 127
TEContinuousStyle routine, 127
TECustomHook routine, 126
TEDrawHook procedure, 126
TEEOLHook procedure, 126
TEFeatureFlag routine, 123–124
TEFindWord procedure, 126–127
teFlushDefault justification mode, 127
teFlushLeft justification mode, 127
teFlushRight justification mode, 127
TEGetPoint routine, 127
TEHitTestHook procedure, 126

TEIdle system call, 125
TEKey routine, 125, 127
Telecommunications. *See* communications
TempDisposeHandle system call, 287–288
TempFreeMem system call, 287–288
TempHLock system call, 288
TempHUnlock system call, 288
TempMaxMem system call, 287–288
TempNewHandle system call, 287–288
Temporary files, 22
Temporary Items folder, 22, 322–323
Temporary memory, 11, 286–288
  compatibility, 287
  system calls, 287–288
TENumStyles routine, 127
TermDataBlock system call, 227
Terminal emulation, 224–230, 232
  tools, 9
Terminal Manager, 9, 210, 213, 217, 227
  handing events with routines, 230
  interfacing with scripting language, 228
  international support, 231
  programming, 224–234
  routines, 228–231
  special keys, 230
  window, 227
Terminal records, 211, 227–228
Terminal tools, 224, 226–228, 230–234
TESetJust routine, 127
TESetStyle routine, 127
TestDeviceAttribute routine, 276
TEWidthHook procedure, 126
Text
  attributes on/off, 127
  buffering, 125–126
  drawing line components, 126
  justifying, 127
  measuring line portions, 126
  mixed-direction, 7
  word breaks, 126
TEXT file format, 94
Text layout services, 340
'TEXT' resource, 173
TextEdit, 121–128
  controlling/checking features, 124
  cursor movement, 124
  customizing, 126
  double-byte characters, 124
  hook procedures, 126–127
  internationalization, 7
  outline highlighting, 124
  Script Manager compatibility, 123
  scripts and primary line direction, 123–124
  synchronizing keyboard and display, 124
  text buffering, 125–126
  two-byte characters, 7
Time
  formats, 139
  measuring intervals, 305–306
Time Manager, 12, 295, 302–306
  accuracy, 303
  compatibility, 304
  fixed-frequency scheduling, 304
  measuring time intervals, 305–306
  task records, 303
  tasks, 304–305
Time-related services, 303
Timekeeping, 12
'tloc' code resource, 232
TMActivate routine, 230
TMAddSearch routine, 229
TMChoose routine, 228
TMClear routine, 229
TMClearSearch routine, 229
TMClick routine, 230
tmClickMsg message, 234
TMCountTermKeys routine, 230
TMDefault routine, 228–229, 231
TMDispose routine, 229
tmDisposeMsg message, 232
TMDoTermKey routine, 230
tmDoTermKeyMsg message, 234
TMEnglishToIntl routine, 231
TMEvent routine, 230
TMGetConfig routine, 228
TMGetCursor routine, 229
TMGetIndTermKey routine, 230
tmGetIndTermKeyMsg message, 234
TMGetProcID system call, 228
TMGetRefCon routine, 231
TMGetSelect routine, 229
TMGetTermEnvirons routine, 230
TMGetTMVersion routine, 213
TMGetToolName routine, 231
TMGetUserData routine, 231
TMGetVersion routine, 231
TMIdle system call, 229
tmInitMsg message, 232
TMIntlToEnglish routine, 231
TMKey routine, 230
tmKeyMsg message, 234
TMMenu routine, 230
TMNew routine, 228
TMRemoveSearch routine, 229
TMReset routine, 229
TMResume routine, 230
TMScroll system call, 229
TMSetConfig routine, 228
TMSetRefCon routine, 231
TMSetSelect routine, 229
TMSetUserData routine, 231
TMStream routine, 229
tmStreamMsg message, 232–233
TMUpdate routine, 230
TMValidate system call, 231
Token Ring, 247
TokenTalk, 247
Tolerant colors, 270
Toolbox Event Manager, 72
Tools
  Correction, 22
  DeRez, 39
  Emulation, 22
  file transfer, 9, 22
  Rez MPW, 39–40
  ScriptCheck MPW, 40
  software installation, 48–49
  terminal emulation, 9
totalLoadCmd sound command, 200–201
Transactions, 155
Translate24To32 system call, 291–292
Transliterate routine, 140

Trash can, 27, 35
Trash folder, 22, 35, 323
TrueType, 5, 32, 60, 103
  B-splines, 110
  characters, 110–111
  contours, 110
  Font Manager, 116–117
  fonts, 6–7, 108–116
  global graphics state, 114
  instructions, 111
  interpreter, 114
  local graphics state, 114
  outlines converted to bitmaps, 114–116
  scaler, 114
  scan converter, 114
TruncText routine, 136
'tscr' code resource, 232
'tset' code resource, 232
Tuples, 242
'tval' code resource, 231
Two-byte characters, 7

U

Unconditional sleep demands, 328
UnholdMemory system call, 285
UnRegisterSection routine, 96
UpdateAlias system call, 316
UpdateGWorld routine, 274–275
UpperText routine, 140
User interface, 13
Users and Groups control panel, 25, 66, 258
Utility synthesizers, 193

V

Vertical Retrace Manager, 303
View menu, 19–20
ViewEdit tool, 334
Virtual memory (VM), 11, 27–28, 60, 281–286
  compatibility, 283–284
  controlling, 284–285
  double page fault, 285
  hard disk space, 15
  locality of reference, 285–286
  pages, 284–285
  requirements, 14–15
VT102 terminal emulation tool, 225–226

W

waitCmd sound command, 200
WaitNextCall routine, 306
WaitNextEvent system call, 59, 60, 72–73, 286–287, 296–298, 300
WakeUpProcess system call, 300
Wave tables, 188, 199
waveTableCmd sound command, 199
WDEF (window definition procedures), 290–291
Window Manager, 72
Window objects help messages, 177–178
Windows
  cache region, 227
  complex, 334
  help messages, 176–177
  help resources, 181–182
  human interface design, 33
  movable object help resources, 183
  palettes, 272–273
  positioning, 33, 72
  sizing, 33
  terminal emulation region, 227
  zoom box, 33
WordBreak procedure, 127
Worldwide Developer's Conference, 3
Write system call, 286
WriteEdition routine, 97
WritePartialResource routine, 313
'wstr' resources, 162

X

XMODEM file transfer tool, 235
XPP driver, 247, 250

Z

ZhongwenTalk script system, 128
ZIP (Zone Information Protocol), 249–250
Zoom box, 33

Other Books Available in the Macintosh Inside Out series

- **Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, Dan Shafer
  Here is the information you need to understand and use the power of MacApp, Apple Computer, Inc.'s official development environment for the Macintosh. The book discusses object-oriented concepts, using MPW with MacApp, the MacApp class library, and creating the Macintosh user interface. All examples are in Apple's Object Pascal language.
  576 pages, paperback
  $24.95, book alone, order number 09784
  $34.95, book/disk, order number 55062

- **C++ Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, Dan Shafer
  In this book you will find information on using MacApp with C++, the up-and-coming language for Macintosh development. The book covers object-oriented techniques, MPW, and the MacApp class libraries. All program examples are in C++.
  600 pages, paperback
  $24.95, book alone, order number 57020
  $34.95, book/disk, order number 57021

- **Elements of C++ Macintosh® Programming**
  Dan Weston
  Macintosh programmers will learn just what they need to take the step from C to C++ programming, the future of Macintosh development. The book covers the basics and then teaches how to design practical programs with C++.
  464 pages, paperback
  $22.95, order number 55025

- **Programmer's Guide to MPW®, Volume I**
  Exploring the Macintosh® Programmer's Workshop
  Mark Andrews
  Learn the secrets to unlocking the power of MPW, Apple's official integrated software development system for the Macintosh. The book begins with fundamental skills and concepts and then progresses to more advanced examples culminating in a fully functional application.
  608 pages, paperback
  $26.95, order number 57011

- **ResEdit™ Complete**
  Peter Alley and Carolyn Strange
  This book/disk package contains the actual ResEdit software along with a complete guide to using it. The book shows you how to customize your desktop and then moves on to cover more advanced topics such as creating standard resources, designing templates, and writing your own resource editor.
  576 pages, paperback
  $29.95 book/disk, order number 55075

- **The Complete Book of HyperTalk® 2**
  Dan Shafer
  This hands-on guide covers HyperTalk 2, with its greatly expanded features and capabilities. It offers practical information on commands, operators, and functions as well as detailed explanations of XCMDs, dialog boxes, menus, communications, and stack design. You'll also find plenty of tips and dozens of ready-to-use scripts.
  480 pages, paperback
  $24.95, order number 57082

▶ **Programming the LaserWriter®**
David A. Holzgang
This practical reference shows how to take advantage of all of the LaserWriter's features and capabilities. Offering numerous useful tips, techniques, and examples, the book takes programmers through the details of accessing the LaserWriter directly and thus bypassing the Apple Printing Manager and its limitations.
464 pages, paperback
$24.95, order number 57068

▶ **Debugging Macintosh® Software with MacsBug®**
Includes MacsBug 6.2
Konstantin Othmer and Jim Straus
This book/disk package is a complete guide to using MacsBug. It includes the actual MacsBug software as well as a hands-on tutorial on using it to debug Macintosh programs. Debugging tips, tricks, and advice appear throughout the book, in addition to numerous examples.
576 pages, paperback
$34.95 book/disk, order number 57049

Order Number	Quantity	Price	Total
________	_____	____	_____
________	_____	____	_____
________	_____	____	_____
________	_____	____	_____

TOTAL ORDER ______

Shipping and state sales tax will be added automatically.

Credit card orders only please.

Offer good in USA only. Prices and availability subject to change without notice.

Name ________________

Address ________________

____

City/State/Zip ________________

Signature (required) ________________

___Visa ___MasterCard ___AmEx

Account # ________________ Exp. Date ________

Addison-Wesley Publishing Company
Order Department
Route 128
Reading, MA 01867
To order by phone, call (800) 477-2226

DATE DUE

OhioLINK			
MAY 01 REC'D			
MAY 03 REC'D			
JUN 12 1996			
JUN 12 REC'D			
DEC 20 1996			
SEP 17 REC'D			
DEC 20 1996			
DEC 17 REC'D			
OhioLINK			
APR 07 REC'D			

DEMCO 38-297

BOWLING GREEN STATE UNIVERSITY DISCARDED LIBRARY

SCI QA 76.76 .O63 M43 1991

Meadow, Anthony.

System 7 revealed

> $22.95 USA
> $29.95 CANADA

System 7 Revealed

ANTHONY MEADOW

"This book is a great first step in tackling 7.0 development."
—*Steve Goldberg, Product Manager, System 7.0, Apple Computer, Inc.*

"A *must* for anyone serious about development on the Macintosh."
—*Art Schumer, Group Manager of Macintosh Technology, Microsoft*

System 7 is the most significant and exciting development in Macintosh® system software since the Macintosh was introduced. It incorporates such impressive advances as virtual memory, inter-application communication, and an enhanced Finder™. Every Macintosh programmer will want to explore and take advantage of System 7's powerful new technology.

System 7 Revealed provides you with a first look at all the features and newest capabilities of System 7. The book describes each feature and function in detail and then shows you how to use the new system calls available for application development. Topics covered include the Memory and Sound Managers, processes, fonts and TrueType™, Publish and Subscribe, File Sharing, and changes to the file system.

You will also learn how to:

- Reduce the complexity of your applications with Apple events and other high-level events
- Communicate with other computers using the Communications Toolbox, Data Access Language, and Apple Data Stream Protocol
- Use the Help Manager to provide on-line help to your users
- Internationalize your software to take advantage of the rapidly growing global demand for Macintosh software

and much more.

This thorough coverage of vital System 7 concepts and features makes **System 7 Revealed** an essential reference for all Macintosh programmers.

Anthony Meadow is cofounder and president of both Bear River Associates, Inc., a leading Macintosh software development company, and Bear River Institute, a premier Macintosh technical training firm. He is also the president of the MacApp® Developers Association, an international organization for commercial and in-house developers using Apple's object-oriented application framework.

ISBN 0-201-55040-7
55040

Cover design by Ronn Campisi
Addison-Wesley Publishing Company, Inc.